ECMA/TC39-TG1/2004/1
Ecma/TC39/2004/4
Draft Standard ECMA-999
February 2004

ECMA International

Standardizing Information and Communication Systems

ECMAScript for XML (E4X)
Specification

Final Draft – 22 February 2004

Phone: +41 22 849.60.00 - Fax: +41 22 849.60.01 - URL: http://www.ecma.ch - Internet: helpdesk@ecma.ch
**Brief History**

On 13 June 2002, a group of companies led by BEA Systems proposed a set of programming language extensions adding native XML support to ECMAScript (ECMA-262). The programming language extensions were designed to provide a simple, familiar, general purpose XML programming model that flattens the XML learning curve by leveraging the existing skills and knowledge of one of the largest developer communities worldwide. The benefits of this XML programming model include reduced code complexity, tighter revision cycles, faster time to market, decreased XML footprint requirements and looser coupling between code and XML data.

The ECMAScript group (ECMA TC39/TG1) unanimously agreed to the proposal and established a sub-group to standardize the syntax and semantics of a general purpose, cross platform, vendor neutral set of programming language extensions called ECMAScript for XML (E4X). The development of this Standard started on 8 August 2002. This Standard was developed as an extension to ECMAScript Edition 3, but may be applied to other versions of ECMAScript as well.

This Standard adds native XML datatypes to the ECMAScript language, extends the semantics of familiar ECMAScript operators for manipulating XML data and adds a small set of new operators for common XML operations, such as searching and filtering. It also adds support for XML literals, namespaces, qualified names and other mechanisms to facilitate XML processing.

This standard will be integrated into future editions of ECMA-262 (ECMAScript). The ECMAScript group is working on significant enhancements for future editions of the ECMAScript language, including mechanisms for defining XML types using the XML Schema language and supports for classes.

This Standard has been adopted as the 1st Edition of ECMA-<TBD ###> by the ECMA General Assembly in June, 2004.

The following people have contributed to this specification:

- John Schneider, BEA/AgileDelta (Lead Editor)
- Jeff Dyer, Macromedia (Supporting Editor)
- Rok Yu, Microsoft (Supporting Editor)
- Steve Adamski, AOL/Netscape
- Patrick Beard, AOL/Netscape
- Adam Bosworth, BEA
- Steve Brandli, BEA
- Vikram Dhaneshwar, Microsoft
- Waldemar Horwat, AOL/Netscape
- Nathaniel Frietas, palmOne
- Gary Grossman, Macromedia
- Ethan Hugg, AgileDelta
- Mark Igra, BEA
- David Jacobs, MITRE
- Alex Khesin, BEA
- Terry Lucas, BEA
- Brent Noorda, Openwave
- Richard Rollman, AgileDelta
- Markus Scherer, IBM
- Michael Shenfield, RIM
- Dan Suciu, University of Washington
- Peter Torr, Microsoft
- Eric Vasilik, BEA
- Wayne Vicknair, IBM
- Herman Venter, Microsoft
- Roger Weber, BEA

*This list may be incomplete. If your name has been omitted, please let me know so I can add it. Also, please provide any additional people (internal or otherwise) who contributed to this specification by providing comments or feedback. Thanks!*
### Table of contents

1  **Scope**  

2  **Status of this Document**  

3  **Normative References**  

4  **Motivation**  
   4.1  The Rise of XML Processing  
   4.2  Current XML Processing Approaches  
   4.2.1  The Document Object Model (DOM)  
   4.2.2  The eXtensible Stylesheet Language (XSLT)  
   4.2.3  Object Mapping  
   4.3  The E4X Approach  

5  **Design Principles**  

6  **Notational Conventions**  
   6.1  **Algorithm Conventions**  
   6.1.1  Indentation Style  
   6.1.2  Property Access  
   6.1.3  Iteration  
   6.1.4  Conditional Repetition  
   6.2  **Method Invocation**  

7  **Lexical Conventions**  
   7.1  Context Keywords  
   7.2  Punctuators  
   7.3  XML Initialiser Input Elements  

8  **Types**  
   8.1  The XML Type  
   8.1.1  Internal Properties and Methods  
   8.1.1.1  [[Get]] (P)  
   8.1.1.2  [[Put]] (P, V)  
   8.1.1.3  [[Delete]] (P)  
   8.1.1.4  [[DeleteByIndex]] (P)  
   8.1.1.5  [[DefaultValue]] (hint)
8.1.1.6 [[HasProperty]] (P) 17
8.1.1.7 [[DeepCopy]] ( ) 17
8.1.1.8 [[Descendants]] (P) 18
8.1.1.9 [[Equals]] (V) 18
8.1.1.10 [[ResolveValue]] ( ) 19
8.1.1.11 [[Insert]] (P, V) 19
8.1.1.12 [[Replace]] (P, V) 20
8.1.1.13 [[AddInScopeNamespace]] ( N ) 20
8.2 The XMLList Type 21
8.2.1 Internal Properties and Methods 21
8.2.1.1 [[Get]] (P) 21
8.2.1.2 [[Put]] (P, V) 22
8.2.1.3 [[Delete]] (P) 24
8.2.1.4 [[DefaultValue]] (hint) 24
8.2.1.5 [[HasProperty]] (P) 24
8.2.1.6 [[Append]] (V) 25
8.2.1.7 [[DeepCopy]] ( ) 25
8.2.1.8 [[Descendants]] (P) 26
8.2.1.9 [[Equals]] (V) 26
8.2.1.10 [[ResolveValue]] ( ) 26
8.3 The AttributeName Type 27
8.3.1 Internal Properties 27
8.4 The AnyName Type 27
9 Type Conversion 28
9.1 ToString 28
9.1.1 ToString Applied to the XML Type 28
9.1.2 ToString Applied to the XMLList Type 29
9.2 ToXMLString ( input argument, [AncestorNamespaces], [IndentLevel] ) 29
9.2.1 ToXMLString Applied to the XML Type 30
9.3 ToXML 32
9.3.1 ToXML Applied to the String Type 32
9.3.2 ToXML Applied to a W3C XML Information Item 33
9.4 ToXMLList 35
9.4.1 ToXMLList Applied to the String Type 36
9.5 ToAttributeName 36
9.5.1 ToAttributeName Applied to the String Type 37
9.6 ToXMLName 37
9.6.1 ToXMLName Applied to the String Type 37
10 Expressions 39
10.1 Primary Expressions 39
10.1.1 Attribute Identifiers 39
10.1.2 Qualified Identifiers 40
10.1.3 Wildcard Identifiers 41
10.1.4 XML Initialiser 42
10.1.5 XMLList Initialiser
10.2 Left-Hand-Side Expressions
10.2.1 Property Accessors
10.2.2 Function Calls
10.2.2.1 CallMethod ( \texttt{r, args} )
10.2.3 XML Descendant Accessor
10.2.4 XML Filtering Predicate Operator
10.3 Unary Operators
10.3.1 The delete Operator
10.3.2 The typeof Operator
10.4 Additive Operators
10.4.1 The Addition Operator ( + )
10.5 Equality Operators
10.5.1 The Abstract Equality Comparison Algorithm
10.6 Assignment Operators
10.6.1 XML Assignment Operator
10.6.2 XMLList Assignment Operator
10.6.3 Compound Assignment (\texttt{op=})

11 Statements
11.1 The default xml namespace Statement
11.1.1 GetDefaultNamespace ( )
11.2 The for-in Statement
11.3 The for-each-in Statement

12 Native E4X Objects
12.1 The Global Object
12.1.1 Internal Properties of the Global Object
12.1.2 Function Properties of the Global Object
12.1.3 Constructor Properties of the Global Object
12.2 Namespace Objects
12.2.1 The Namespace Constructor Called as a Function
12.2.2 The Namespace Constructor
12.2.3 Properties of the Namespace Constructor
12.2.4 Properties of the Namespace Prototype Object (Built-in Methods)
12.2.5 Properties of Namespace Instances
12.3 QName Objects
12.3.1 The QName Constructor Called as a Function
12.3.2 The QName Constructor
12.3.3 Properties of the QName Constructor
12.3.4 Properties of the QName Prototype Object
12.3.5 Properties of QName Instances
12.4 XML Objects
12.4.1 The XML Constructor Called as a Function
12.4.2 The XML Constructor
12.4.3 Properties of the XML Constructor
12.4.3.7  XML.settings ( )  74
12.4.3.8  XML.setSettings ( [ Settings ] )  74
12.4.3.9  XML.defaultSettings ( )  75
12.4.3.10  [[HasInstance]] ( V )  75
12.4.4  Properties of the XML Prototype Object (Built-in Methods)  76
12.4.5  Properties of XML Instances  89
12.5  XMLList Objects  89
12.5.1  The XMLList Constructor Called as a Function  89
12.5.2  The XMLList Constructor  89
12.5.3  Properties of the XMLList Constructor  89
12.5.3.1  XMLList.prototype  89
12.5.4  Properties of the XMLList Prototype Object (Built-in Methods)  90

13  Errors  96

A  Optional Features (Non-normative)  97
A.1  XML Built-in Methods  97
A.2  XMLList Built-in Methods  98
1 Scope
This standard defines the syntax and semantics of ECMAScript for XML (E4X), a set of programming language extensions adding native XML support to ECMAScript.

2 Status of this Document
This document is a final draft produced to motivate, facilitate and finalize discussions related to E4X with the goal of creating a general purpose, cross platform, vendor neutral E4X standard. It is considered stable, but has not yet been agreed or otherwise sanctioned by ECMA. Comments and suggestions are solicited and encouraged.

3 Normative References
Document Object Model (DOM) Level 2 Specifications, W3C Recommendation, 13 November 2000


Extensible Markup Language 1.0 (Second Edition), W3C Recommendation 6 October 2000

Namespaces in XML, W3C Recommendation, 14 January 1999

ISO/IEC 10646-1:1993 Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus its amendments and corrigenda.


XML Information Set, W3C Recommendation 24 October 2001

XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999


4 Motivation

4.1 The Rise of XML Processing

Developing software to create, navigate and manipulate XML data is a significant part of every developer’s job. Developers are inundated with data encoded in the eXtensible Markup Language (XML). Web pages are increasingly encoded using XML vocabularies, including XHTML and Scalable Vector Graphics (SVG). On mobile devices, data is encoded using the Wireless Markup Language (WML). Web services interact using the Simple Object Access Protocol (SOAP) and are described using the Web Service Description Language (WSDL). Deployment descriptors, project make files and configuration files and now encoded in XML, not to mention an endless list of custom XML vocabularies designed for vertical industries. XML data itself is even described and processed using XML in the form of XML Schemas and XSL Stylesheets.

4.2 Current XML Processing Approaches

Current XML processing techniques require ECMAScript programmers to learn and master a complex array of new concepts and programming techniques. The XML programming models often seem heavyweight, complex and unfamiliar for ECMAScript programmers. This section provides a brief overview of the more popular XML processing techniques.

4.2.1 The Document Object Model (DOM)

One of the most common approaches to processing XML is to use a software package that implements the interfaces defined by the W3C XML DOM (Document Object Model). The XML DOM represents XML data using a general purpose tree abstraction and provides a tree-based API for navigating and manipulating the data (e.g., getParentNode(), getChildNodes(), removeChild(), etc.).

This method of accessing and manipulating data structures is very different from the methods used to access and manipulate native ECMAScript data structures. ECMAScript programmers must learn to write tree navigation algorithms instead of object navigation algorithms. In addition, they have to learn a relatively complex interface hierarchy for interacting with the XML DOM. The resulting XML DOM code is generally harder to read, write, and maintain than code that manipulates native ECMAScript data structures. It is more verbose and often obscures the developer’s intent with lengthy tree navigation logic. Consequently, XML DOM programs require more time, knowledge and resources to develop.

4.2.2 The eXtensible Stylesheet Language (XSLT)

XSLT is a language for transforming XML documents into other XML documents. Like the XML DOM, it represents XML data using a tree-based abstraction, but also provides an expression language called XPath designed for navigating trees. On top of this, it adds a declarative, rule-based language for matching portions of the input document and generating the output document accordingly.

From this description, it is clear that XSLT’s methods for accessing and manipulating data structures are completely different from those used to access and manipulate ECMAScript data structures. Consequently, the XSLT learning curve for ECMAScript programmers is quite steep. In addition to learning a new data model, ECMAScript programmers have to learn a declarative programming model, recursive decent processing model, new expression language, new XML language syntax, and a variety of new programming concepts (templates, patterns, priority rules, etc.). These differences also make XSLT code harder to read, write and maintain for the ECMAScript programmer. In addition, it is not possible to use familiar development environments, debuggers and testing tools with XSLT.

4.2.3 Object Mapping

Several have also tried to navigate and manipulate XML data by mapping it to and from native ECMAScript objects. The idea is to map XML data onto a set of ECMAScript objects, manipulate those objects directly, then map them back to XML. This allows ECMAScript programmers to reuse their knowledge of ECMAScript objects to manipulate XML data.

This is a great idea, but unfortunately it does not work for a wide range of XML processing tasks. Native ECMAScript objects do not preserve the order of the original XML data and order is significant for XML. Not only do XML developers need to preserve the order of XML data, but they also need to control and manipulate the order of XML data. In addition, XML data contains artifacts that are not easily represented by the ECMAScript object model, such as namespaces, attributes, comments, processing instructions and mixed element content.
4.3 The E4X Approach

ECMAScript for XML was envisioned to address these problems. E4X extends the ECMAScript object model with native support for XML data. It reuses familiar ECMAScript operators for creating, navigating and manipulating XML, such that anyone who has used ECMAScript is able to start using XML with little or no additional knowledge. The extensions include native XML data types, XML literals (i.e., initialisers) and a small set of new operators useful for common XML operations, such as searching and filtering.

E4X applications are smaller and more intuitive to ECMAScript developers than comparable XSLT or DOM applications. They are easier to read, write and maintain requiring less developer time, skill and specialized knowledge. The net result is reduced code complexity, tighter revision cycles and shorter time to market for Internet applications. In addition, E4X is a lighter weight technology enabling a wide range of mobile applications.
5 Design Principles

The following design principles are used to guide the development of E4X and encourage consistent design decisions. They are listed here to provide insight into the E4X design rational and to anchor discussions on desirable E4X traits.

- **Simple:** One of the most important objectives of E4X is to simplify common programming tasks. Simplicity should not be compromised for interesting or unique features that do not address common programming problems.

- **Consistent:** The design of E4X should be internally consistent such that developers can anticipate its behaviour.

- **Familiar:** Common operators available for manipulating ECMAScript objects should also be available for manipulating XML data. The semantics of the operators should not be surprising to those familiar with ECMAScript objects. Developers already familiar with ECMAScript objects should be able to begin using XML objects with minimal surprises.

- **Minimal:** Where appropriate, E4X defines new operators for manipulating XML that are not currently available for manipulating ECMAScript objects. This set of operators should be kept to a minimum to avoid unnecessary complexity. It is a non-goal of E4X to provide, for example, the full functionality of XPath.

- **Loose Coupling:** To the degree practical, E4X operators will enable applications to minimize their dependencies on external data formats. For example, E4X applications should be able to extract a value deeply nested within an XML structure, without specifying the full path to the data. Thus, changes in the containment hierarchy of the data will not require changes to the application.

- **Complementary:** E4X should integrate well with other languages designed for manipulating XML, such as XPath, XSLT and XML Query. For example, E4X should be able to invoke complementary languages when additional expressive power is needed without compromising the simplicity of the E4X language itself.
6 Notational Conventions

This specification extends the notational conventions used in the ECMAScript Edition 3 specification. In particular, it extends the algorithm notation to improve the clarity, readability and maintainability of this specification. The new algorithm conventions are described in this section.

6.1 Algorithm Conventions

This section introduces the algorithm conventions this specification adds to those used to describe the semantics of ECMAScript Edition 3. These conventions are not part of the E4X language. They are used within this specification to describe the semantics of E4X operations.

6.1.1 Indentation Style

This specification extends the notation used in the ECMAScript Edition 3 specification by defining an algorithm indentation style. The new algorithm indentation style is used in this specification to group related collections of steps together. This convention is useful for expressing a set of steps that are taken conditionally or repeatedly. For example, the following algorithm fragment uses indentation to describe a set of steps that are taken conditionally:

1. If `resetParmeters` is true
   a. Let `x = 0`
   b. Let `y = 0`
   c. Let `deltaX = 0.5`
2. Else
   a. Let `deltaX = deltaX + accelerationX`

In the example above, steps 1.a through 1.c are taken if the condition expressed in step 1 evaluates to true. Otherwise, step 2.a is taken.

Standard outline numbering form is used to identify steps and distinguish nested levels of indentation when it might not otherwise be obvious due to pagination.

6.1.2 Property Access

This specification extends the notation used in the ECMAScript Edition 3 specification by defining three property access conventions. When used on the left hand side of an assignment operation in this specification, the property access conventions are used to modify the value of a specified property of a specified object. In other contexts in this specification, the property access conventions are used for specifying that the value of a specified property be retrieved from a specified object based on its property name.

There are three forms of the property access conventions, two for accessing normal properties and one for accessing internal properties. The first convention for accessing normal properties is expressed using the following notation:

```
object . propertyName
```

When used on the left hand side of an assignment operation, this property access convention is equivalent to calling the `[[Put]]` method of `object`, passing the string literal containing the same sequence of parameters as `propertyName` and the value from the right hand side of the assignment operator as arguments. For example, the following algorithm fragment:

```
1. Let `item.price = "5.95"
```

is equivalent to the following algorithm fragment:

```
1. Call the `[[Put]]` method of `item` with arguments "price" and "5.95"
```

When used in other contexts, this property access convention is equivalent to calling the `[[Get]]` method of `object` passing the string literal containing the same sequence of characters as `propertyName` as an argument. For example, the following algorithm fragment:
1. Let \( \text{currentPrice} = \text{item.price} \)

is equivalent to the following algorithm fragment:

1. Let \( \text{currentPrice} \) be the result of calling the \([[\text{Get}]]\) method of \( \text{item} \) with argument "\text{price}".

The second convention for accessing normal properties is expressed using the following notation:

\[
\text{object} \ [ \ \text{propertyName} \ ]
\]

When used on the left hand side of an assignment operation, this property access convention is equivalent to calling the \( \text{Object.\{\[\text{Put}\]\}} \) method with \( \text{object} \) as the \text{this} object, passing \( \text{ToString(\text{propertyName})} \) and the value from the right hand side of the assignment operator as arguments. For example, the following algorithm fragment:

1. Let \( \text{item}[1] = \text{item2} \)

is equivalent to the following algorithm fragment:

1. Call the \( \text{Object.\{\[\text{Put}\]\}} \) method with \( \text{item} \) as the \text{this} object and arguments \( \text{ToString(1)} \) and \( \text{item2} \)

When used in other contexts, this property access convention is equivalent to calling the \( \text{Object.\{\[\text{Get}\]\}} \) method with \( \text{object} \) as the \text{this} object and argument \( \text{ToString(\text{propertyName})} \). For example, the following algorithm fragment:

1. Let \( \text{item2} = \text{item}[1] \)

is equivalent to the following algorithm fragment:

1. Let \( \text{item2} \) be the result of calling the \( \text{Object.\{\[\text{Get}\]\}} \) method with \( \text{item} \) as the \text{this} object and argument \( \text{ToString(1)} \)

This is a convenient and familiar notation for specifying numeric property names used as array indices.

The convention for accessing internal property names, including those that refer to internal methods, is specified using the following notation:

\[
\text{object} \ . \ [[ \ \text{internalPropertyName} \ ]]
\]

When used on the left hand side of an assignment operation, this property access convention is equivalent to setting the value of the \( [[\text{internalPropertyName}]] \) of the specified \( \text{object} \) to the value from the right hand side of the assignment operator. For example, the following algorithm fragment:

1. Let \( x.[[\text{Class}]] = \text{"element"} \)

is equivalent to the following algorithm fragment:

1. Let the value of the \( [[\text{Class}]] \) property of \( x \) be "element"

When used in other contexts, this property access convention is equivalent to getting the value of the \( [[\text{internalPropertyName}]] \) property of \( \text{object} \). For example, the following algorithm fragment:

1. Let \( \text{class} = x.[[\text{Class}]] \)

is equivalent to the following algorithm fragment:

1. Let \( \text{class} \) be the value of the \( [[\text{Class}]] \) property of \( x \).
6.1.3 Iteration

This specification extends the notation used for describing ECMAScript Edition 3 by defining two iteration conventions. These iteration conventions are used by this specification for expressing that a set of steps must be taken once for each item in a collection or once for each integer in a specified range.

The first iteration convention is defined for expressing a sequence of steps that must be taken once for each member of a collection. It is expressed using the following for each notation:

For each item in collection steps

This for each notation is equivalent to performing the given steps repeatedly with the variable item bound to each member of collection. The order in which item is bound to members of collection is implementation dependent. The repetition ends after item has been bound to all the members of collection or when the algorithm exits via a return or a thrown exception. The steps may be specified on the same line following a comma or on the following lines using the indentation style described in section 6.1.1.

For example,

1. Let total = 0
2. For each product in groceryList
   a. If product.price > maxPrice, throw an exception
   b. Let total = total + product.price

In this example, steps 2.a and 2.b are repeated once for each member of the collection groceryList or until an exception is thrown in line 2.a. The variable product is bound to the value of a different member of groceryList before each repetition of these steps.

The second iteration convention defined by this specification is for expressing a sequence of steps that must be repeated once for each integer in a specified range of integers. It is expressed using the following for notation:

For variable = first to last steps

This for notation is equivalent to computing first and last, which will evaluate to integers i and j respectively, and performing the given steps repeatedly with the variable variable bound to each member of the sequence i, i+1 … j in numerical order. The values of first and last are computed once prior to performing steps and do not change while performing steps. The repetition ends after variable has been bound to each item of this sequence or when the algorithm exits via a return or a thrown exception. The steps may be specified on the same line following a comma or on the following lines using the indentation style described above. For example,

1. For i = 0 to priceList.length-1, call ToString(priceList[i])

In this example, ToString is called once for each item in priceList in sequential order.

A modified version of the for notation exists for iterating through a range of integers in reverse sequential order. It is expressed using the following notation:

For variable = first downto last steps

The modified for notation works exactly as described above except the variable variable is bound to each member of the sequence i, i-1, .. j in reverse numerical order. If i is less than j, the steps are not performed.

6.1.4 Conditional Repetition

This specification extends the notation used in the ECMAScript Edition 3 specification by defining a convention for expressing conditional repetition of a set of steps. This convention is defined by the following notation:

While ( expression ) steps
The while notation is equivalent to computing the expression, which will evaluate to either true or false and if it is true, taking the given steps and repeating this process until the expression evaluates to false or the algorithm exits via a return or a thrown exception. The steps may be specified on the same line following a comma or on the following lines using the indentation style described above. For example,

1. Let \( \log_2 = 0 \)
2. While \((n > 1)\)
   a. Let \( n = n / 2 \)
   b. Let \( \log_2 = \log_2 + 1 \)

In this example, steps 2.a and 2.b are repeated until the expression \( n > 1 \) evaluates to false.

### 6.2 Method Invocation

This specification extends the notation used in the ECMAScript Edition 3 specification by defining a method invocation convention. The method invocation convention is used in this specification for calling a method of a given object passing a given set of arguments and returning the result. This convention is defined by the following notation:

\[
\text{object} . \text{methodName} ( \text{arguments} )
\]

where \text{arguments} is a comma separated list of zero or more values. The method invocation notation is equivalent to constructing a new Reference \( r \) with base object set to \( \text{object} \) and property name set to a string literal containing the same sequence of characters as \text{methodName}, constructing an internal list \( l \) of the values in \text{arguments}, invoking the CallMethod operator (section 10.2.2.1) passing \( r \) and \( l \) as arguments and returning the result. For example, the following algorithm fragment:

1. Let \( sub = s . \text{substring}(2, 5) \)

Is equivalent to the following algorithm fragment:

1. Let \( r \) be a new Reference with base object = \( s \) and property name = "substring"
2. Let \( l \) be an internal list containing the values 2 and 5
3. Let \( sub = \text{CallMethod}(r, l) \)
7 Lexical Conventions

This section introduces the lexical conventions E4X adds to ECMAScript.

E4X modifies the existing lexical grammar productions for `InputElementRegExp` and `Punctuators`. It also introduces the goal symbols `InputElementXMLTag` and `InputElementXMLContent` that describe how sequences of Unicode characters are translated into parts of XML initialisers.

In addition to the grammar contexts where a division (/) or division-assignment (/=) is permitted, the `InputElementDiv` symbol is used in those syntactic grammar contexts where a less than (<), less than or equals (<=), left shift (<<) or left shift-assignment (<<=) operator is permitted. The `InputElementXMLTag` is used in those syntactic contexts where the literal contents of an XML tag are permitted. The `InputElementXMLContent` is used in those syntactic contexts where the literal contents of an XML element are permitted. The `InputElementRegExp` symbol is used in all other syntactic grammar contexts.

The addition of the production `InputElementRegExp :: XMLMarkup` and extended use of the existing production `InputElementRegExp :: Punctuator :: <` allow the start of XML initialisers to be identified.

To better understand when these goal symbols apply, consider the following example:

```
order = <{x}>{item}</{x}>;
```

The input elements returned from the lexical grammar along with the goal symbol and productions used for this example are as follows:

<table>
<thead>
<tr>
<th>Input Element</th>
<th>Goal</th>
<th>Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td><code>InputElementRegExp</code></td>
<td><code>Token::Identifier</code></td>
</tr>
<tr>
<td>=</td>
<td><code>InputElementDiv</code></td>
<td><code>Punctuator</code></td>
</tr>
<tr>
<td>&lt;</td>
<td><code>InputElementRegExp</code></td>
<td><code>Punctuator</code></td>
</tr>
<tr>
<td>{</td>
<td><code>InputElementXMLTag</code></td>
<td><code>{'</code></td>
</tr>
<tr>
<td>x</td>
<td><code>InputElementRegExp</code></td>
<td><code>Token::Identifier</code></td>
</tr>
<tr>
<td>}</td>
<td><code>InputElementDiv</code></td>
<td><code>Punctuator</code></td>
</tr>
<tr>
<td>&gt;</td>
<td><code>InputElementXMLTag</code></td>
<td><code>XMLTagPunctuator</code></td>
</tr>
<tr>
<td>}</td>
<td><code>InputElementXMLContent</code></td>
<td><code>{</code></td>
</tr>
<tr>
<td>item</td>
<td><code>InputElementRegExp</code></td>
<td><code>Token::Identifier</code></td>
</tr>
<tr>
<td>}</td>
<td><code>InputElementDiv</code></td>
<td><code>Punctuator</code></td>
</tr>
<tr>
<td>&lt;/</td>
<td><code>InputElementXMLContent</code></td>
<td><code>&lt;/</code></td>
</tr>
<tr>
<td>{</td>
<td><code>InputElementXMLTag</code></td>
<td><code>{'</code></td>
</tr>
<tr>
<td>x</td>
<td><code>InputElementRegExp</code></td>
<td><code>Token::Identifier</code></td>
</tr>
<tr>
<td>}</td>
<td><code>InputElementDiv</code></td>
<td><code>Punctuator</code></td>
</tr>
<tr>
<td>&gt;</td>
<td><code>InputElementRegExp</code></td>
<td><code>Token::Identifier</code></td>
</tr>
<tr>
<td>;</td>
<td><code>InputElementXMLElement</code></td>
<td><code>XMLTagPunctuator</code></td>
</tr>
</tbody>
</table>

Syntax

E4X extends the `InputElementRegExp` goal symbol defined by ECMAScript with the following production:

```
InputElementRegExp ::
    XMLMarkup
```

E4X extends ECMAScript by adding the following goal symbols:

```
InputElementXMLElement ::
    XMLTagCharacters
XMLTagPunctuator
XMLAttributeValue
{
```
7.1 Context Keywords
E4X extends ECMAScript by adding a set of context keywords. Context keywords take on a specific meaning when used in specified contexts where identifiers are not permitted by the syntactic grammar. However, they differ from ECMAScript Edition 3 keywords in that they may also be used as identifiers. E4X does not add any additional keywords to ECMAScript.

Syntax
E4X extends ECMAScript by replacing the `Identifier` production and adding a `ContextKeyword` production as follows:

```plaintext
Identifier ::= IdentifierName but not ReservedWord or ContextKeyword
               ContextKeyword

ContextKeyword ::= each
               xml
               namespace
```

7.2 Punctuators
E4X extends the list of Punctuators defined by ECMAScript by adding the descendent (`..`) input element to support the XML descendent accessor (section 10.2.3), the attribute (@) input element to support XML attribute lookup (section 10.1.1) and the name qualifier (:) input element to support qualified name lookup (section 10.1.2). E4X extends the `Punctuator` non-terminal with the following production:

```plaintext
Punctuator ::= .. @ ::
```

7.3 XML Initialiser Input Elements
The goal symbols `InputElementXMLTag` and `InputElementXMLContent` describe how Unicode characters are translated into input elements that describe parts of XML initialisers. These input elements are consumed by the syntactic grammars described in sections 10.1.4 and 10.1.5.

The lexical grammar allows characters which may not form a valid XML initialiser. The syntax and semantics described in the syntactic grammar ensure that the final initialiser is well formed XML.

Unlike in string literals, the back slash (`\`) is not treated as the start of an escape sequence inside XML initialisers. Instead the XML entity references specified in the XML 1.0 specification should be used to escape characters. For example, the entity reference `&apos;` can be used for a single quote (‘), `&quot;` for a double quote ("), and `&lt;` for less than (<).

The left curly brace (`{`) and right curly brace (`}`) are used to delimit expressions that may be embedded in tags or element content to dynamically compute portions of the XML initialiser. The curly braces may appear in literal form inside an attribute value, a CDATA, PI, or XML Comment. In all other cases, the character reference `&#x7B;` must be used to represent the left curly brace (`{`) and the character reference `&#x7D;` must be used to represent the right curly brace (`}`).

Syntax
XMLMarkup ::
  XMLComment
  XMLCDATA
  XMLPI

XMLTagCharacters ::
  SourceCharacters but no embedded XMLTagPunctuator
  or left-curly { or quote ' or double-quote " or forward-slash / or XMLWhitespace

XMLWhitespace ::
  <SP>
  <TAB>
  <CR>
  <LF>

XMLText ::
  SourceCharacters but no embedded left-curly { or less-than <

XMLComment ::
  <!-- XMLCommentCharacters_opt -->

XMLCommentCharacters ::
  SourceCharacters but no embedded sequence --

XMLCDATA ::
  <![CDATA[ XMLCDATACharacters_opt ]]>
8 Types
E4X extends ECMAScript by adding two new fundamental data types for representing XML elements and XML lists. Future versions will also provide the capability to derive user-defined types for specific XML vocabularies using XML Schemas.

8.1 The XML Type
The XML type is an ordered collection of properties with a name, a set of XML attributes, a set of in-scope namespaces and a parent. Each property of an XML object has a unique numeric property name \( P \), such that ToUint32(ToUint32(\( P \))) is equal to \( P \), and has a value of type XML representing a child node. The name of an XML object is a QName object or null. Each XML attribute is an instance of the XML type. Each namespace is a Namespace object. The parent is a value of type XML or null. Methods are associated with XML objects using non-numeric property names.

Each value of type XML represents an XML element, attribute, comment, processing-instruction or text node. The internal \([\text{Class}]\) property is set to “element”, “attribute”, “comment”, “processing-instruction” or “text” as appropriate. Each XML object representing an XML attribute, comment, processing-instruction (PI) or text node has no user visible properties and stores a String value representing the value of the associated attribute, comment, PI or text node in the \([\text{Value}]\) property logically inherited from the Object type.

E4X intentionally blurs the distinction between an individual XML object and an XMLList containing only that object. To this end, all operations available for XMLList objects are also available for XML objects. Implementations that extend E4X should preserve this constraint.

Note: The internal XML data model described above represents XML child nodes as properties with numeric property names. The numeric names of these properties indicate the ordinal position of a given child within its parent. The values of these properties are XML objects that have an associated name (e.g., an element name). E4X defines XML \([\text{Get}]\) and \([\text{Put}]\) operators (below) that provide access to the properties of an XML object based on the names of the property values rather than their internal numeric property names.

8.1.1 Internal Properties and Methods
Internal properties and methods are not part of the E4X language. They are defined by this specification purely for expository purposes. An implementation of E4X must behave as if it produced and operated upon internal properties in the manner described here. This specification reuses the notation for internal properties from the ECMAScript Edition 3 specification, wherein the names of internal properties are enclosed in double square brackets \([ [ ] ]\). When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a TypeError exception is thrown.

The XML type is logically derived from the Object type and inherits its internal properties. Unless otherwise specified, the XML type also inherits the type conversion semantics defined for the Object type (section 9 of ECMAScript Edition 3). The following table summarises the internal properties the XML type adds to those defined by the Object type.

<table>
<thead>
<tr>
<th>Property</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Name]]</td>
<td>None</td>
<td>The name of this XML object.</td>
</tr>
<tr>
<td>[[Parent]]</td>
<td>None</td>
<td>The parent of this XML object.</td>
</tr>
<tr>
<td>[[Attributes]]</td>
<td>None</td>
<td>The attributes associated with this XML object.</td>
</tr>
<tr>
<td>[[InScopeNamespaces]]</td>
<td>None</td>
<td>The namespaces in scope for this XML object.</td>
</tr>
<tr>
<td>[[Length]]</td>
<td>None</td>
<td>The number of ordered properties in this XML object.</td>
</tr>
<tr>
<td>[[DeleteByIndex]]</td>
<td>(PropertyName)</td>
<td>Deletes a property with the numeric index PropertyName</td>
</tr>
<tr>
<td>[[DeepCopy]]</td>
<td>()</td>
<td>Returns a deep copy of this XML object.</td>
</tr>
<tr>
<td>[[ResolveValue]]</td>
<td>()</td>
<td>Returns this XML object. This method is used when attempting to resolve the value of an empty XMLList.</td>
</tr>
<tr>
<td>[[Descendants]]</td>
<td>(PropertyName)</td>
<td>Returns an XMLList containing the descendants of this XML object with names that match propertyName.</td>
</tr>
<tr>
<td>[[Equals]]</td>
<td>(Value)</td>
<td>Returns a boolean value indicating whether this XML object has the same XML content as the given XML Value.</td>
</tr>
<tr>
<td>[[Insert]]</td>
<td>(PropertyName, Value)</td>
<td>Inserts one or more new properties before the property with name PropertyName (a numeric index).</td>
</tr>
<tr>
<td>[[Replace]]</td>
<td>(PropertyName, Value)</td>
<td>Replaces the value of the property with name PropertyName (a</td>
</tr>
</tbody>
</table>
The value of the `[[Name]]` property must be `null` or a QName object containing a legal XML element name, attribute name, or PI name. The value of the `[[Name]]` property is `null` if and only if the XML object represents an XML comment or text node. The `[[Name]]` for each XML object representing a processing-instruction will have its `uri` property set to the empty string.

The value of the `[[Parent]]` property must be either an XML object or `null`. When an XML object occurs as a property (i.e., a child) of another XML object, the `[[Parent]]` property is set to the containing XML object (i.e., the parent).

The value of the `[[Attributes]]` property is a set of zero or more XML objects. When a new object is added to the `[[Attributes]]` set, it replaces any existing object in `[[Attributes]]` that has the same set identity. The set identity of each XML object `x` in `[[Attributes]]` is defined to be `x.[[Name]]`. Therefore, there exists no two objects `x, y` in `[[Attributes]]` such that the result of the comparison `x.[[Name]] == y.[[Name]]` is `true`. The value of the `[[Attributes]]` property is the empty set if the XML object represents an XML attribute, comment, PI or text node. Note: Although namespaces are declared using attribute syntax in XML, they are not represented in the `[[Attributes]]` property.

The value of the `[[InScopeNamespaces]]` property is a set of zero or more Namespace objects representing the namespace declarations in scope for this XML object. When a new object is added to the `[[InScopeNamespaces]]` set, it replaces any existing object in the `[[InScopeNamespaces]]` set that has the same set identity. The set identity of each Namespace object `n` in `[[InScopeNamespaces]]` for which `n.prefix` is not `undefined`, is defined to be `n.prefix`. The set identity of each Namespace object `n` in `[[InScopeNamespaces]]` for which `n.prefix` is `undefined` is unique. Therefore, there exists no two objects `x, y` in `[[InScopeNamespaces]]`, such that the result of the comparison `x.prefix == y.prefix` is `true` unless the result of the comparison `x.prefix == undefined` is `true`.

The value of the `[[Length]]` property is a non-negative integer.

### 8.1.1.1 `[[Get]]` (P)

**Overview**

The XML type overrides the internal `[[Get]]` method defined by the Object type. The XML `[[Get]]` method is used to retrieve an XML attribute by its name or a set of XML elements by their names. The input argument `P` may be an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input argument `P` is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument `P` is an unqualified XML attribute name, it identifies XML attributes in no namespace.

In addition, the input argument `P` may be a numeric property name. If `P` is a numeric property name, the XML `[[Get]]` method converts this XML object to an `XMLList l` and calls the `[[Get]]` method of `l` with argument `P`. This treatment intentionally blurs the distinction between a single XML object and an `XMLList` containing only one value.

Note: Unlike the internal Object. `[[Get]]` method, the internal XML `[[Get]]` method is never used for retrieving methods associated with XML objects. E4X modifies the ECMAScript method lookup semantics for XML objects as described in section 10.2.2.

**Semantics**

When the `[[Get]]` method of an XML object `x` is called with property name `P`, the following steps are taken:

1. If `ToString(ToInt32(P)) == P`
   a. Let `l = ToXMLList(x)`
   b. Return the result of calling the `[[Get]]` method of `l` with argument `P`
2. Let `n = ToXMLName(P)`
3. Let `l` be a new `XMLList with l.([TargetObject]) = x` and `l.([TargetProperty]) = n`
4. If `Type(n)` is `AttributeName`
   a. For each `a in x.([Attributes])`
When the `[[Put]]` method of an XML object takes parameters `P` and `V`, the following steps are taken:

1. If `(Type(V) \notin \{XML, XMLList\})` or `(V.[[Class]] \in \{"text", "attribute"\})`
   a. Let `c = ToString(V)`
2. Else
   a. Let `c` be the result of calling the `[[DeepCopy]]` method of `V`
3. If `ToString(ToUint32(P)) !== P`, throw a `TypeError` exception
   
   Note: This operation is reserved for future versions of E4X.
4. If `x.[[Class]] \in \{"text", "comment", "processing-instruction", "attribute"\}`, return
5. Let `n = ToXMLName(P)`
6. Let `defaultNamespace = GetDefaultNamespace()`
7. If `Type(n)` is `AttributeName`
   a. Call the function `isXMLName` (section 12.1.2.1) with argument `n.[[Name]]` and if the result is `false`, return
   b. If `Type(c)` is `XMLList`
      i. If `c.[[Length]] === 0`, let `c` be the empty string
      ii. Else
         1. Let `s = ToString(c[0])`
         2. For `i = 1` to `c.[[Length]].1`
            a. Let `s` be the result of concatenating `s`, the string " " (space) and `ToString(c[i])`
         3. Let `c = s`
   c. Else
      i. Let `c = ToString(c)`
    d. Let `a = null`
  e. For `j = 0` to `x.[[Attributes]].length`
     i. If `(n.[[Name]].localName === x.[[Attributes]][j].[[Name]].localName)` and `(n.[[Name]].uri === null)` or `(n.[[Name]].uri === x.[[Attributes]][j].[[Name]].uri)`
        1. If `(a === null)`, let `a = x.[[Attributes]][j]`
2. Else call the [[Delete]] method of \( x \) with argument \( x.\text{[Attributes]}[j].\text{Name} \)

f. If \( a \) == null
   i. If \( n.\text{[Name]}\).\text{uri} == null
      1. Let \( \text{nons} \) be a new Namespace created as if by calling the constructor new Namespace()
      2. Let \( \text{name} \) be a new QName created as if by calling the constructor new QName(\( \text{nons} \),
         \( n.\text{[Name]} \))
   ii. else
      1. Let \( \text{name} \) be a new QName created as if by calling the constructor new QName(\( n.\text{[Name]} \))
   iii. Create a new XML object \( a \) with \( a.\text{[Name]} = \text{name} \), \( a.\text{[Class]} = \text{"attribute"} \) and \( a.\text{[Parent]} = x \)
   iv. Let \( x.\text{[XMLAttributes]} = x.\text{[XMLAttributes]} \cup \{ a \} \)
   v. Let \( \text{ns} \) be the result of calling the [[GetNamespace]] method of \( \text{name} \) with no arguments
   vi. Call the [[AddInScopeNamespace]] method of \( x \) with argument \( \text{ns} \)
   g. Let \( a.\text{[Value]} = c \)
   h. Return

8. Call the function isXMLName (section 12.1.2.1) with argument \( n \) and if the result is false, return

9. Let \( i = \text{undefined} \)

10. Let \( \text{primitiveAssign} = \text{(Type}(c) \neq \{ \text{XML}, \text{XMLList} \}) \) and \( \text{(n.localName is not equal to the string "*"}) \)

11. For \( (k = 0 \) to \( x.\text{[Length]} - 1) \)
    a. If \( (n.\text{localName} == \text{"*"})
       or \( (x[k].\text{[Class]} == \text{"element"}) \) and \( (x[k].\text{[Name]},\text{localName}=n.\text{localName}) \)
       and \( ((n.\text{uri} == \text{null}) \) or \( (n.\text{uri} == x[k].\text{[Name]},\text{uri}) \)
          i. If \( i == \text{undefined} \), let \( i = k \)
          ii. Else call the [[DeleteByIndex]] property of \( x[k] \) with argument ToString(\( k \))
    b. If \( \text{primitiveAssign} == \text{true} \)
       i. If \( (n.\text{uri} == \text{null}) \)
          1. Let \( \text{name} \) be a new QName created as if by calling the constructor new QName(\( \text{defaultNamespace}, n \))
          ii. Else
             1. Let \( \text{name} \) be a new QName created as if by calling the constructor new QName(\( n \))
             iii. Create a new XML object \( y \) with \( y.\text{[Name]} = \text{name} \), \( y.\text{[Class]} = \text{"element"} \) and \( y.\text{[Parent]} = x \)
             iv. Let \( \text{ns} \) be the result of calling [[GetNamespace]] on \( \text{name} \) with no arguments
             v. Call the [[Replace]] method of \( x \) with arguments ToString(\( i \)) and \( y \)
             vi. Call the [[AddInScopeNamespace]] on \( y \) with argument \( \text{ns} \)
    c. If \( s \) is not the empty string, call the [[Replace]] method of \( x[i] \) with arguments "0" and \( s \)
12. If \( i == \text{undefined} \)
    a. Delete all the properties of the XML object \( x[i] \)
    b. Let \( s = \text{ToString}(c) \)
    c. If \( s \) is not the empty string, call the [[Replace]] method of \( x[i] \) with arguments "0" and \( s \)
13. Else
    a. Call the [[Replace]] method of \( x \) with arguments ToString(\( i \)) and \( c \)
14. Return

8.1.1.3  [[Delete]] (P)

Overview

The XML type overrides the internal [[Delete]] method defined by the Object type. The XML [[Delete]] method is used to remove a set of XML attributes by name or a set of XML valued properties by name. Unlike, the internal Object [[Delete]], the XML [[Delete]] method shifts all the properties following deleted properties up to fill in empty slots created by the delete. The input argument \( P \) may be an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading "@" symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard "*" or the attributes wildcard "@*". When the input argument \( P \) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument \( P \) is an unqualified XML attribute name, it identifies XML attributes in no namespace.
In addition, the input argument \( P \) may be a numeric property name. If \( P \) is a numeric property name, the XML \([\text{Get}]\) method converts this XML object to an XMLList \( l \) and calls the \([\text{Delete}]\) method of \( l \) with argument \( P \). This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value.

**Semantics**

When the \([\text{Delete}]\) method of an XML object \( x \) is called with property name \( P \), the following steps are taken:

1. If ToUint32(ToUInt32(P)) == P
   a. Let \( l = \text{ToXMLList}(x) \)
   b. Return the result of calling the \([\text{Delete}]\) method of \( l \) with argument \( P \)
2. Let \( n = \text{ToXMLName}(P) \)
3. If Type(\( n \)) is AttributeName
   a. For each \( a \) in \( x.\text{Attributes} \)
      - If (\( n.\text{Name}.localName == "*" \)
        and \( (n.\text{Name}.uri == \text{null}) \)
      i. Let \( a.\text{Parent} = \text{null} \)
      ii. Remove the attribute \( a \) from \( x.\text{Attributes} \)
   b. Return true
4. Let \( dp = 0 \)
5. For each property \( q \) in \( x \)
   a. If \( (n.\text{Name} == "*" \)
        or \( x[q].\text{Class} == \text{"element"} \)
        and \( (n.\text{uri} == \text{null}) \)
      i. Let \( q.\text{Parent} = \text{null} \)
      ii. Remove the property \( q \) from \( x \)
      iii. Let \( dp = dp + 1 \)
   b. Else
      i. If \( dp > 0 \), rename \( q \) to ToUInt32(ToUInt32(\( q \)) – \( dp \))
6. Let \( x.\text{Length} = x.\text{Length} - dp \)
7. Return true.

**8.1.1.4 \([\text{DeleteByIndex}]\) (P)**

**Overview**

The XML type adds the internal \([\text{DeleteByIndex}]\) method to the methods defined by the Object type. The XML \([\text{DeleteByIndex}]\) method is used to remove an XML property by its numeric property name. Like, the XML \([\text{Delete}]\) method, the XML \([\text{DeleteByIndex}]\) method shifts all the properties following deleted properties up to fill in empty slots created by the delete. The input argument \( P \) may be a numeric property name.

**Semantics**

When the \([\text{DeleteByIndex}]\) method of an XML object \( x \) is called with property name \( P \), the following steps are taken:

1. Let \( i = \text{ToUInt32}(P) \)
2. If ToUInt32(\( i \)) == P
   a. If \( i \) is greater than or equal to \( x.\text{Length} \), return true
   b. Else
      i. Let \( x[P].\text{Parent} = \text{null} \)
      ii. Remove the property with the name \( P \) from \( x \)
      iii. For each property \( q \) of \( x \) such that ToUInt32(\( q \)) > \( i \), rename \( q \) to ToUInt32(ToUInt32(\( q \)) – 1)
      iv. Let \( x.\text{Length} = x.\text{Length} - 1 \)
   c. Return true
3. else throw a TypeError exception

**8.1.1.5 \([\text{DefaultValue}]\) (hint)**

**Overview**

The XML type adds the internal \([\text{DefaultValue}]\) method to the methods defined by the Object type. The XML \([\text{DefaultValue}]\) method sets the value of a property to a default value when the property is not defined. The input argument \( P \) may be a numeric property name.

**Semantics**

When the \([\text{DefaultValue}]\) method of an XML object \( x \) is called with property name \( P \), the following steps are taken:

1. Let \( i = \text{ToUInt32}(P) \)
2. If ToUInt32(\( i \)) == P
   a. If \( i \) is greater than or equal to \( x.\text{Length} \), return true
   b. Else
      i. Let \( x[P].\text{Parent} = \text{null} \)
      ii. Remove the property with the name \( P \) from \( x \)
      iii. For each property \( q \) of \( x \) such that ToUInt32(\( q \)) > \( i \), rename \( q \) to ToUInt32(ToUInt32(\( q \)) – 1)
      iv. Let \( x.\text{Length} = x.\text{Length} - 1 \)
   c. Return true
3. else throw a TypeError exception
The XML type overrides the internal [[DefaultValue]] method defined by the Object type. The XML [[DefaultValue]] method returns a primitive value representing this XML object. Unlike, the [[DefaultValue]] method defined by the Object type, the XML [[DefaultValue]] method always returns a string. The hint parameter is ignored.

**Semantics**

When the [[DefaultValue]] method of an XML object \(x\) is called with parameter \(hint\), the following step is taken:

1. \(\text{Return } \text{ToString}(x)\)

8.1.1.6 [[HasProperty]] (P)

**Overview**

The XML type overrides the internal [[HasProperty]] method defined by the Object type. The XML [[HasProperty]] method is used to determine whether this XML object contains an XML element or attribute by its name. The input argument \(P\) may be an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading "@" symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard "*" or the attributes wildcard "@*". When the input argument \(P\) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument \(P\) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

In addition, the input argument \(P\) may be a numeric property name. If \(P\) is a numeric property name equal to "0", the XML [[HasProperty]] method returns true. If \(P\) is a numeric property name other than "0", the XML [[HasProperty]] method returns false. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value.

**Semantics**

When the [[HasProperty]] method of an XML object \(x\) is called with property name \(P\), the following steps are taken:

1. If \(\text{ToString}(\text{ToUint32}(P)) == P\)
   a. \(\text{Return } (P == "0")\)
2. Let \(n = \text{ToXMLName}(P)\)
3. If \(\text{Type}(n)\) is AttributeName
   a. For each \(a\) in \(x.\text{[Attributes]}\)
      i. If \((n.\text{[Name]}.\text{localName} == "*"\) or \((n.\text{[Name]}.\text{localName} == a.\text{[Name]}.\text{localName})\)
         and \((n.\text{[Name]}.\text{uri} == \text{null})\) or \((n.\text{[Name]}.\text{uri} == a.\text{[Name]}.\text{uri})\)
            i. \(\text{Return } \text{true}\)
   b. \(\text{Return } \text{false}\)
4. For \((k = 0 \text{ to } x.\text{[Length]}-1)\)
   a. If \((n.\text{localName} == "*"\)
      or \((x[k].\text{[Class]} == "element")\) and \((x[k].\text{[Name]}.\text{localName} == n.\text{localName}))\)
      and \((n.\text{uri} == \text{null})\) or \((n.\text{uri} == x[k].\text{[Name]}.\text{uri})\)
         i. \(\text{Return } \text{true}\)
5. \(\text{Return } \text{false}\)

8.1.1.7 [[DeepCopy]] ( )

**Overview**

The XML type adds the internal [[DeepCopy]] method to the internal properties defined by the Object type. The XML [[DeepCopy]] method is used to create and return a deep copy of this object, including its attributes, properties, namespaces and the attributes, properties and namespaces of all its descendants. The internal [[Parent]] property of the return value is set to null and the internal [[Parent]] property of each copied descendant is set to its newly copied parent as appropriate.

**Semantics**
When the [[DeepCopy]] method of an XML object \( x \) is called, the following steps are taken:

1. Let \( y \) be a new XML object
2. For each internal property \([p]\) of \( x \), except \([\text{Parent}]\), \([\text{Attributes}]\) and \([\text{InScopeNamespaces}]\)
   a. Let \( y.\[p] \) be a copy of \( x.\[p] \)
3. For each \( ns \) in \( x.\[\text{InScopeNamespaces}]\)
   a. Let \( ns2 \) be a new Namespace created as if by calling the constructor \( \text{new Namespace}(ns.\text{prefix}, ns.\text{uri}) \)
   b. Let \( y.\[\text{InScopeNamespaces}] = y.\[\text{InScopeNamespaces}] \cup \{ ns2 \} \)
4. Let \( y.\[\text{Parent}] = \text{null} \)
5. For each attribute \( a \) in \( x.\[\text{Attributes}]\)
   a. Let \( b \) be the result of calling the [[DeepCopy]] method of \( a \)
   b. Let \( b.\[\text{Parent}] = y \)
   c. Let \( y.\[\text{Attributes}] = y.\[\text{Attributes}] \cup \{ b \} \)
6. For \( i = 0 \) to \( x.\[\text{Length}] - 1 \)
   a. Let \( c \) be the result of calling the [[DeepCopy]] method of \( x[i] \)
   b. Let \( y[i] = c \)
   c. Let \( y.\[\text{Parent}] = y \)
7. Return \( y \)

### 8.1.1.8 [[Descendants]] (P)

**Overview**

The XML type adds the internal [[Descendants]] method to the internal properties defined by the Object type. The XML [[Descendants]] method is used to retrieve all the XML valued descendants of this XML object (i.e., children, grandchildren, great-grandchildren, etc.) with names matching the input argument \( P \). The input argument \( P \) may be an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input argument \( P \) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument \( P \) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

**Semantics**

When the [[Descendants]] method of an XML object \( x \) is called with property name \( P \), the following steps are taken:

1. Let \( n = \text{ToXMLName}(P) \)
2. Let \( l \) be a new XMLList with \( l.\[\text{TargetObject}] = \text{null} \)
3. If \( \text{Type}(n) \) is AttributeName
   a. For each attribute \( a \) in \( x.\[\text{Attributes}]\)
      i. If \((n.[\text{Name}].\text{localName} = "*"\)) or \((n.[\text{Name}].\text{localName} = a.[\text{Name}].\text{localName})\) and \((n.[\text{Name}].\text{uri} = \text{null})\) or \((n.[\text{Name}].\text{uri} = a.[\text{Name}].\text{uri})\)
         1. Call the [[Append]] method of \( l \) with argument \( a \)
4. For \( k = 0 \) to \( x.\[\text{Length}] - 1 \)
   a. If \((n.\text{localName} = "*"\))
      or \((x[k].\[\text{Class}] = "element")\) and \((x[k].\[\text{Name}].\text{localName} = n.\text{localName})\)
      and \((n.\text{uri} = \text{null})\) or \((n.\text{uri} = x[k].\[\text{Name}].\text{uri})\)
         i. Call the [[Append]] method of \( l \) with argument \( x[k] \)
   b. Let \( dq \) be the resultsof calling the [[Descendants]] method of \( x[k] \) with argument \( P \)
   c. If \( dq.\[\text{Length}] > 0 \), call the [[Append]] method of \( l \) with argument \( dq \)
5. Return \( l \)

### 8.1.1.9 [[Equals]] (V)

**Overview**

The XML type adds the internal [[Equals]] method to the internal properties defined by the Object type. The XML [[Equals]] method is used to compare this XML object for XML content equality with another XML object \( V \). The [[Equals]] operator returns \text{true} if \( V \) is a value of type XML considered equal to this XML object. Otherwise, it returns \text{false}. 
Semantics
When the \[\text{[Equals]}\] method of an XML object \(x\) is called with value \(V\), the following steps are taken:

1. If \(\text{Type}(V)\) is not XML, return \text{false}
2. If \(x.\text{[Class]}\) is not equal to \(V.\text{[Class]}\), return \text{false}
3. If \(x.\text{[Name]}\) is not \text{null}
   a. If \(x.\text{[Name]}.\text{localName}\) is not equal to \(V.\text{[Name]}.\text{localName}\), return \text{false}
   b. If \(x.\text{[Name]}.\text{uri}\) is not equal to \(V.\text{[Name]}.\text{uri}\), return \text{false}
4. If \(x.\text{[Attributes]}\) does not contain the same number of items as \(V.\text{[Attributes]}\), return \text{false}
5. If \(x.\text{[Length]}\) is not equal to \(V.\text{[Length]}\), return \text{false}
6. If \(x.\text{[Value]}\) is not equal to \(V.\text{[Value]}\), return \text{false}
7. For each attribute \(a\) in \(x.\text{[Attributes]}\)
   a. If \(V.\text{[Attributes]}\) does not contain an attribute \(b\), such that \(b.\text{[Name]}.\text{localName} == a.\text{[Name]}.\text{localName},\)
   \(b.\text{[Name]}.\text{uri} == a.\text{[Name]}.\text{uri}\) and \(b.\text{[Value]} == a.\text{[Value]}\), return \text{false}
8. For \(i = 0\) to \(x.\text{[Length]}-1\)
   a. Let \(r\) be the result of calling the \[\text{[Equals]}\] method of \(x[i]\) with argument \(V[i]\)
   b. If \(r == \text{false}\), return \text{false}
9. Return \text{true}

8.1.1.10 \[\text{[ResolveValue]}\] ( )
Overview
The XML type adds the internal \[\text{[ResolveValue]}\] method to the internal properties defined by the Object type. The XML \[\text{[ResolveValue]}\] method returns this XML object. It is used by the XMLList \[\text{[ResolveValue]}\] method to support \[\text{[Put]}\] operations on empty XMLLists.

Semantics
When the \[\text{[ResolveValue]}\] method of an XML object \(x\) is called, the following step is taken:

1. Return \(x\)

8.1.1.11 \[\text{[Insert]}\] ( \(P, V\) )
Overview
The XML type adds the internal \[\text{[Insert]}\] method to the internal properties defined by the Object type. The XML \[\text{[Insert]}\] method is used to insert a value \(V\) at a specific position \(P\). The input argument \(P\) must be a numeric property name. The input argument \(V\) may be a value of type XML, XMLList or any value that can be converted to a String with ToString().

Semantics
When the \[\text{[Insert]}\] method of an XML object \(x\) is called with property name \(P\) and value \(V\), the following steps are taken:

1. If \(x.\text{[Class]}\) \(\in\) \{"text", "comment", "processing-instruction", "attribute"\}, return
2. Let \(i = \text{ToUint32}(P)\)
3. If (ToString\((i)\) is not equal to \(P\)), throw a \text{TypeError} exception
4. Let \(n = 1\)
5. If \(\text{Type}(V)\) is XMLList, let \(n = V.\text{[Length]}\)
6. If \(n == 0\), Return
7. For \(j = x.\text{[Length]}-1\) downto \(i\), rename property ToString\((j)\) of \(x\) to ToString\((j + n)\)
8. Let \(x.\text{[Length]} = x.\text{[Length]} + n\)
9. If \(\text{Type}(V)\) is XMLList
   a. For \(j = 0\) to \(V.\text{[Length]}-1\), \(x[i+j] = V[j]\)
10. Else
    a. Call the \[\text{[Replace]}\] method of \(x\) with arguments \(i\) and \(V\)
11. Return
8.1.1.12 [[Replace]] (P, V)

**Overview**

The XML type adds the internal `[[Replace]]` method to the internal properties defined by the Object type. The XML `[[Replace]]` method may be used to replace the property at a specific position $P$ with the value $V$. The input argument $P$ must be a numeric property name. The input argument $V$ may be a value of type XML, XMLList or any value that can be converted to a String with `ToString()`.

**Semantics**

When the `[[Replace]]` method of an XML object $x$ is called with property name $P$ and value $V$, the following steps are taken:

1. If $x.\text{[[Class]]} \in \{"text", "comment", "processing-instruction", "attribute"\}$, return
2. Let $i = \text{ToUint32}(P)$
3. If $(\text{ToString}(i)$ is not equal to $P)$, throw a `TypeError` exception
4. If $i$ is greater than or equal to $x.\text{[[Length]]}$,
   a. Let $P = \text{ToString}(x.\text{[[Length]]})$
   b. Let $x.\text{[[Length]]} = x.\text{[[Length]]} + 1$
5. If Type($V$) is XML and $V.\text{[[Class]]} \in \{"element", "comment", "processing-instruction", "text"\}$
   a. Let $V.\text{[[Parent]]} = x$
   b. Let $x[P].\text{[[Parent]]} = \text{null}$
   c. Let $x[P] = V$
   Note: The E4X data model does not enforce the constraint: $\forall x \in \text{XML} : x.\text{[[InScopeNamespaces]]} \supseteq x.\text{[[Parent]]}.\text{[[InScopeNamespaces]]}$. However, implementations may at this point add namespaces from $V.\text{[[InScopeNamespaces]]}$ to $x$ or any ancestors of $x$. Likewise, implementations may at this point add namespaces from $x$ to $V$ or any descendents of $V$.
6. Else if Type($V$) is XMLList
   a. Call the `[[DeleteByIndex]]` method of $x$ with argument $P$
   b. Call the `[[Insert]]` method of $x$ with arguments $P$ and $V$
7. Else
   a. Let $s = \text{ToString}(V)$
   b. Create a new XML object $t$ with $t.\text{[[Class]]} = "text"$, $t.\text{[[Parent]]} = x$ and $t.\text{[[Value]]} = s$
   c. Let $x[P].\text{[[Parent]]} = \text{null}$
   d. Let the value of property $P$ of $x$ be $t$
8. Return

8.1.1.13 [[AddInScopeNamespace]] (N)

**Overview**

The XML type adds the internal `[[AddInScopeNamespace]]` method to the internal properties defined by the Object type. The XML `[[AddInScopeNamespace]]` method is used to add a new Namespace to the `[[InScopeNamespaces]]` of a given XML object. The input argument $N$ is a value of type Namespace to be added to the `[[InScopeNamespaces]]` property of this XML object.

**Semantics**

When the `[[AddInScopeNamespace]]` method of an XML object $x$ is called with a namespace $N$, the following steps are taken:

1. If $x.\text{[[Class]]} \in \{"text", "comment", "processing-instruction", "attribute"\}$, return
2. If $N.\text{prefix} === \text{undefined}$
   a. Let $match = \text{null}$
   b. For each $ns$ in $x.\text{[[InScopeNamespaces]]}$
      i. If $ns.\text{uri} === N.\text{uri}$, let $match = ns$
   c. If $match === \text{null}$,
i. Let $x.\text{[InScopeNamespaces]} = x.\text{[InScopeNamespaces]} \cup \{ N \}$

3. else
   a. if $N.\prefix = ""$ and $x.\text{[Name]}\text{.uri} = ""$, return
   b. Let $\text{match}$ be $\text{null}$
   c. For each $ns$ in $x.\text{[InScopeNamespaces]}$
      i. If $N.\prefix = ns.\prefix$, let $\text{match} = ns$
   d. If $\text{match}$ is not $\text{null}$ and $\text{match}.\text{uri}$ is not equal to $N.\text{uri}$, let $\text{match}.\prefix = \text{undefined}$
   e. Let $x.\text{[InScopeNamespaces]} = x.\text{[InScopeNamespaces]} \cup \{ N \}$

4. Note: The E4X data model does not enforce the constraint: $\forall x \in \text{XML} : x.\text{[InScopeNamespaces]} \supseteq x.\text{[Parent]}\text{[InScopeNamespaces]}$. However, implementations may at this point add $N$ to the $\text{[InScopeNamespaces]}$ property of any ancestors or descendents of $x$.

5. Return

8.2 The XMLList Type

The XMLList type is an ordered collection of properties. Each property of an XMLList object has a unique numeric property name $P$, such that $\text{ToString(ToUint32(P))}$ is equal to $P$ and a value of type XML. Methods are associated with XMLList objects using non-numeric property names.

A value of type XMLList represents an XML document, XML fragment or an arbitrary collection of XML objects (e.g., a query result).

E4X intentionally blurs the distinction between an individual XML object and an XMLList containing only that object. To this end, all operations available for XML objects are also available for XMLList objects. Implementations that extend E4X should preserve this constraint.

8.2.1 Internal Properties and Methods

The XMLList type is logically derived from the Object type and inherits its internal properties. Unless otherwise specified, the XMLList type also inherits the type conversion semantics defined for the Object type (section 9 of ECMA Script Edition 3). The following table summarises the internal properties the XMLList type adds to those defined by the Object type.

<table>
<thead>
<tr>
<th>Property</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Length]]</td>
<td>None</td>
<td>The number of properties contained in this XMLList object.</td>
</tr>
<tr>
<td>[[TargetObject]]</td>
<td>None</td>
<td>The XML or XMLList object associated with this object that will be affected when items are inserted into this XMLList.</td>
</tr>
<tr>
<td>[[TargetProperty]]</td>
<td>None</td>
<td>The name of a property that may be created in the [[TargetObject]] when objects are added to an empty XMLList.</td>
</tr>
<tr>
<td>[[Append]]</td>
<td>(Value)</td>
<td>Appends a new property to the end of this XMLList object.</td>
</tr>
<tr>
<td>[[DeepCopy]]</td>
<td>()</td>
<td>Returns a deep copy of this XMLList object.</td>
</tr>
<tr>
<td>[[Descendants]]</td>
<td>(PropertyName)</td>
<td>Returns an XMLList containing all the descendants of values of in this XMLList that have names matching propertyName.</td>
</tr>
<tr>
<td>[[Equals]]</td>
<td>(Value)</td>
<td>Returns a Boolean value indicating whether this XMLList object has the same content as the given Value or this XMLList object contains an object that compares equal to the given Value.</td>
</tr>
<tr>
<td>[[ResolveValue]]</td>
<td>()</td>
<td>Resolves the value of this XML object. If this XML object is not empty, it is returned. Otherwise, [[ResolveValue]] attempts to create an appropriate value.</td>
</tr>
</tbody>
</table>

The value of the [[Length]] property is a non-negative Number.

8.2.1.1 [[Get]] (P)

Overview

The XMLList type overrides the internal [[Get]] method defined by the Object type. The XMLList [[Get]] method is used to retrieve a specific property of this XMLList object by its numeric property name or to iterate over the XML valued properties.
of this XMLList object retrieving their XML attributes by name or their XML valued properties by name. The input argument $P$ may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input argument $P$ is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument $P$ is an unqualified XML attribute name, it identifies XML attributes in no namespace. Note: Unlike the internal Object. [[Get]] method, the internal XMLList [[Get]] method is never used for retrieving methods associated with XMLList objects. E4X modifies the ECMAScript method lookup semantics for XMLList objects as described in section 10.2.2.

### Semantics

When the [[Get]] method of an XMLList object $x$ is called with property name $P$, the following steps are taken:

1. If ToString(ToUint32($P$)) === $P$
   a. Return the result of calling the Object. [[Get]] method with $x$ as the this object and argument $P$
2. Let $l$ be a new XMLList with $l$. [[TargetObject]] = $x$ and $l$. [[TargetProperty]] = $P$
3. For $i = 0$ to $x$. [[Length]] - 1,
   a. If $x[i]$. [[Class]] === "element",
      i. Let $gq$ be the result of calling the [[Get]] method of $x[i]$ with argument $P$
      ii. If $gq$. [[Length]] > 0, call the [[Append]] method of $l$ with argument $gq$
4. Return $l$

#### 8.2.1.2 [[Put]] ($P$, $V$)

### Overview

The XMLList type overrides the internal [[Put]] method defined by the Object type. The XMLList [[Put]] method is used to modify or replace an XML object within the XMLList and the context of its parent. In addition, when the XMLList contains a single property with an XML object, the [[Put]] method is used to modify, replace, and insert properties or XML attributes of that value by name. The input argument $P$ identifies which portion of the XMLList and associated XML objects will be affected and may be a numeric property name, an unqualified name for an XML attribute (distinguished from XML valued property names by a leading “@” symbol) or set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes or the properties wildcard “*”. When the input argument $P$ is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument $P$ is an unqualified XML attribute name, it identifies XML attributes in no namespace. The input argument $V$ may be a value of type XML, XMLList or any value that can be converted to a String with ToString().

Note: Unlike the internal Object. [[Put]] method, the internal XMLList [[Put]] method is never used for modifying the set of methods associated with XMLList objects.

### Semantics

When the [[Put]] method of an XMLList object $x$ is called with property name $P$ and value $V$, the following steps are taken:

1. Let $i$ = ToUint32($P$)
2. If ToString($i$) === $P$
   a. If $x$. [[TargetObject]] is not null
      i. Let $r$ be the result of calling the [[ResolveValue]] method of $x$. [[TargetObject]]
      ii. If $r$ === null, return
   b. Else let $r = null$
   c. If $i$ is greater than or equal to $x$. [[Length]]
      i. If Type($r$) is XMLList
         1. If $r$. [[Length]] is not equal to 1, return
         2. Else let $r = r[0]$
      ii. Create a new XML object $y$ with $y$. [[Parent]] = $r$, $y$. [[Name]] = $x$. [[TargetProperty]],
          $y$. [[Attributes]] = {}, $y$. [[Length]] = 0
iii. If `Type(x.["TargetProperty"])` is `AttributeName`
   1. Let `attributeExists` be the result of calling the `[[Get]]` method of `r` with argument `y.["Name"]`
   2. If `attributeExists` `[[Length]]` > 0), return
   3. Let `y.["Class"]` = "attribute"
iv. Else if `x.["TargetProperty"]` == `null` or `x.["TargetProperty"].localName == "*"`
   1. Let `y.["Name"]` = `null`
   2. Let `y.["Class"]` = "text"
   v. Else let `y.["Class"]` = "element"
   vi. Let `i = x.["Length"]`
   vii. If `y.["Class"]` is not equal to "attribute")
      1. If `y.["Parent"]` is not `null`
         a. If `(i > 0)`
            i. Let `j = 0`
            ii. While `(j < y.["Parent"].[[Length]]-1)` and `(y.["Parent"][j]` is not the same object as `x[i-1])`
            1. Let `j = j + 1`
         b. Else
            i. Let `j = y.["Parent"].[[Length]]-1`
            ii. Call the `[[Replace]]` method of `y.["Parent"]] with arguments `ToString(j+i) and y`
            2. If `Type(V)` is `XML`, let `y.["Name"] = V.["Name"]`
            3. else if `Type(V)` is `XMLList`, let `y.["Name"] = V.["PropertyName"]`
   viii. Call the `[[Append]]` method of `x` with argument `y`
   d. If `Type(V) \notin \{XML, XMLList\}` or `(V.["Class"] \in \{"text", "attribute")`, let `V = ToString(V)`
   e. If `x[i].[[Class]]` == "attribute"
      i. Call the `[[Put]]` method of `x[i].["Parent"]] with arguments `x[i].["Name"]` and `V`
   ii. Let `attr` be the result of calling `[[Get]]` on `x[i].["Parent"]] with argument `x[i].["Name"]`
   iii. Let `x[i] = attr[0]`
   f. Else if `Type(V)` is `XMLList`
      i. Create a shallow copy `c` of `V`
      ii. Let `parent = x[i].["Parent"]`
      iii. If `parent` is not `null`
            1. Let `q` be the property of `parent`, such that `parent[q]` is the same object as `x[i]`
            2. Call the `[[Replace]]` method of `parent` with arguments `q and c`
            3. For `j = 0` to `c.["Length"]-1`
               a. Let `c[j] = parent[ToString32(q)+j]`
   iv. For `j = x.["Length"]-1 downto i`, rename property `j` of `x` to `ToString(j + c.["Length"])`
   v. For `j = 0` to `c.["Length"]-1`, let `x[i+j] = c[j]`
   vi. Let `x.["Length"] = x.["Length"] + c.["Length"]`
   g. Else if `Type(V)` is `XML` or `(x[i].["Class"] \in \{"text", "comment", "processing-instruction")`)
      i. Let `parent = x[i].["Parent"]`
      ii. If `parent` is not `null`
            1. Let `q` be the property of `parent`, such that `parent[q]` is the same object as `x[i]`
            2. Call the `[[Replace]]` method of `parent` with arguments `q and V`
            3. Let `V = parent[q]`
      iii. Let `x[i] = V`
   h. Else
      i. Call the `[[Put]]` method of `x[i]` with arguments "*" and `V`
   3. Else if `x.["Length"]` is less than or equal to 1
      a. If `x.["Length"]` == 0
         i. Let `r be the result of calling the [[ResolveValue]] method of `x`
         ii. If `(r == null)` or `(r.["Length"]` is not equal to 1), return
         iii. Call the `[[Append]]` method of `x` with argument `r`
      b. Call the `[[Put]]` method of `x[0]` with arguments `P` and `V`
   4. Return
8.2.1.3  [[Delete]] (P)

Overview

The XMLList type overrides the internal [[Delete]] method defined by the Object type. The XMLList [[Delete]] method is used to remove a specific property of the XMLList by its numeric property name or to iterate over the XML valued properties of the XMLList removing their XML attributes or elements by name. The input argument \( P \) may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input argument \( P \) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument \( P \) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

Semantics

When the [[Delete]] method of an XMLList object \( x \) is called with property name \( P \), the following steps are taken:

1. Let \( i = \text{ToUint32}(P) \)
2. If \( \text{ToString}(i) \equiv P \)
   a. If \( i \) is greater than or equal to \( x.\text{[Length]} \), return \text{true}
   b. Else
      i. Let \( \text{parent} = x[i].\text{[Parent]} \)
      ii. If \( \text{parent} \) is not \text{null}
         1. If \( x[i].\text{[Class]} \equiv \text{"attribute"} \)
            a. Call the [[Delete]] method of \( \text{parent} \) with argument \( x[i].\text{[Name]} \)
         2. Else
            a. Let \( q \) be the property of \( \text{parent} \), where \( \text{parent}[q] \) is the same object as \( x[i] \)
            b. Call the [[DeleteByIndex]] method of \( \text{parent} \) with argument \( q \)
      iii. Remove the property with the name \( P \) from \( x \)
   iv. For each property \( q \) in \( x \) such that \( \text{ToUint32}(q) > i \), rename \( q \) to \( \text{ToString}(\text{ToUint32}(q) - 1) \)
   v. Let \( x.\text{[Length]} = x.\text{[Length]} - 1 \)
   c. Return \text{true}
3. For each property \( q \) in \( x \),
   a. If \( x[q].\text{[Class]} \equiv \text{"element"} \)
      i. Call the [[Delete]] method of \( x[q] \) with argument \( P \)
4. Return \text{true}

8.2.1.4  [[DefaultValue]] (hint)

Overview

The XMLList type overrides the internal [[DefaultValue]] method defined by the Object type. The XMLList [[DefaultValue]] method returns a primitive value representing this XMLList object. Unlike, the Object [[DefaultValue]] method, the XMLList [[DefaultValue]] method always returns a string. The hint parameter is ignored.

Semantics

When the [[DefaultValue]] method of an XMLList object \( l \) is called with parameter \( \text{hint} \), the following step is taken:

1. Return \( \text{ToString}(l) \)

8.2.1.5  [[HasProperty]] (P)

Overview

The XMLList type overrides the internal [[HasProperty]] method defined by the Object type. The XMLList [[HasProperty]] method is used to determine whether this XMLList object contains an XML element or attribute by its ordinal position or whether any of the objects contained in this XMLList object contains an XML element or attribute by its name. The input argument \( P \) may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of
XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName
for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input argument P is an
unqualified XML element name, it identifies XML elements in the default namespace. When the input argument P is an
unqualified XML attribute name, it identifies XML attributes in no namespace.

Semantics

When the [[HasProperty]] method of an XMLList object x is called with property name P, the following steps are taken:

1. If ToString(ToUint32(P)) == P
   a. Return (ToUint32(P) < x.[[Length]])
2. For i = 0 to x.[[Length]]-1
   a. If x[i].[[Class]] == "element" and the result of calling the [[HasProperty]] method of x[i] with argument P == true, return true
3. Return false

8.2.1.6  [[Append]] (V)

Overview

The XMLList type adds the internal [[Append]] method to the internal properties defined by the Object type. The XMLList
[[Append]] method is used to append zero or more values specified by V to the end of the XMLList. The input argument V
must be a value of type XMLList or XML.

Semantics

When the [[Append]] method of an XMLList object x is called with value V, the following steps are taken:

1. Let i = x.[[Length]]
2. Let n = 1
3. If Type(V) is XMLList,
   a. Let x.[[TargetObject]] = V.[[TargetObject]]
   b. If x.[[TargetProperty]] is not equal to V.[[TargetProperty]], let x.[[TargetProperty]] = null
   c. Let n = V.[[Length]]
   d. If n == 0, Return
   e. For j = 0 to V.[[Length]]-1, let x[i+j] = V[j]
4. Else [Note: Type(V) is XML]
   a. Let x.[[TargetObject]] = V.[[Parent]]
   b. If x.[[Class]] == "processing-instruction", let x.[[TargetProperty]] = null
   c. Else if x.[[TargetProperty]] is not equal to V.[[Name]], let x.[[TargetProperty]] = null
   d. Let x[i] = V
5. Let x.[[Length]] = x.[[Length]] + n
6. Return

8.2.1.7  [[DeepCopy]] ( )

Overview

The XMLList type adds the internal [[DeepCopy]] method to the internal properties defined by the Object type. The XMLList
[[DeepCopy]] method is used to create and return a copy of this XMLList object containing deep copies of all its properties.

Semantics

When the [[DeepCopy]] method of an XMLList object x is called the following steps are taken:

1. Let l be a new XMLList object
2. Copy all internal properties of x to l
3. For i = 0 to x.[[Length]]-1
   a. Let l[i] be the result of calling the [[DeepCopy]] method of x[i]
Return $l$

### 8.2.1.8 [[Descendants]] ($P$)

**Overview**

The XMLList type adds the internal [[Descendants]] method to the internal properties defined by the Object type. The XMLList [[Descendants]] method may be used to retrieve all the XML valued descendants of the properties in this XMLList (i.e., children, grandchildren, great-grandchildren, etc.) with names matching the input argument $P$. The input argument $P$ may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input argument $P$ is an unqualified XML element name, it identifies XML elements in the default namespace. When the input argument $P$ is an unqualified XML attribute name, it identifies XML attributes in no namespace.

**Semantics**

When the [[Descendants]] method of an XML object $x$ is called with property name $P$, the following steps are taken:

1. Let $l$ be a new XMLList with $l.[[TargetObject]] = \text{null}$
2. For each property $q$ in $x$
   a. If ($x[q].[[Class]] = \text{"element"}$)
      i. Let $dq$ be the result of calling the [[Descendants]] method of $x[q]$ with argument $P$
      ii. If $dq.[[Length]] > 0$, call the [[Append]] method of $l$ with argument $dq$
3. Return $l$

### 8.2.1.9 [[Equals]] ($V$)

**Overview**

The XMLList type adds the internal [[Equals]] method to the internal properties defined by the Object type. The XMLList [[Equals]] method is used to compare this XMLList object for content equality with another XMLList object $V$ or determine whether this XMLList object contains a single XML object that compares equal to $V$. The [[Equals]] operator returns true if this XMLList object is considered equal to $V$ or contains a single XML object considered equal to $V$. Otherwise, it returns false. Empty XMLList objects are considered equal to undefined. The input argument $V$ may be a value of type XMLList, XML, undefined or any value that can be converted to a String with ToString().

**Semantics**

When the [[Equals]] method of an XML object $x$ is called with value $V$, the following steps are taken:

1. If $V == \text{undefined}$ and $x.[[Length]] == 0$, return true
2. If Type($V$) is XMLList
   a. If $V.[[Length]] == 1$, call the [[Equals]] method of $x$ with argument $V[0]$ and return the result
   b. If $x.[[Length]] == 1$, call the [[Equals]] method of $V$ with argument $x[0]$ and return the result
   c. If $x.[[Length]]$ is not equal to $V.[[Length]]$, return false
   d. For $i = 0$ to $x.[[Length]]$
      i. If the result of the comparison $x[i] == V[i]$ is false, return false
      e. Return true
3. Return false

### 8.2.1.10 [[ResolveValue]] ()

**Overview**

The XMLList type adds the internal [[ResolveValue]] method to the internal properties defined by the Object type. The XMLList [[ResolveValue]] method is used to resolve the value of empty XMLLists. If this XMLList object is not empty, the [[ResolveValue]] method will return it. If this XMLList is empty, the [[ResolveValue]] method will attempt to create it based on the [[TargetObject]] and [[TargetProperty]] properties. If the XMLList cannot be created, [[ResolveValue]] returns null.
Semantics

When the `[[ResolveValue]]` method of an XMLList object `x` is called, the following steps are taken:

1. If `x.([[Length]]) > 0`, return `x`
2. Else
   a. If `(x.([[TargetObject]]) == null)` or `(type(x.([[TargetProperty]])) is AttributeName) or (x.([[TargetProperty]]) == null)` or `(x.([[TargetProperty]]).localName == "*")`
      i. Return `null`
   b. Let `base` be the result of calling the `[[ResolveValue]]` method of `x.([[TargetObject]])` recursively
   c. If `base == null`, return `null`
   d. Let `target` be the result of calling `[[Get]]` on `base` with argument `x.([[TargetProperty]])`
   e. If `(target.([[Length]]) == 0)`
      i. If (Type(`base`) is XMLList) and `(base.([[Length]]) > 1)`, return `null`
      ii. Call `[[Put]]` on `base` with arguments `x.([[TargetProperty]])` and the empty string
      iii. Let `target` be the result of calling `[[Get]]` on `base` with argument `x.([[TargetProperty]])`
   f. Return `target`

8.3 The `AttributeName` Type

The internal `AttributeName` type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of E4X must behave as if it produced and operated upon AttributeNames in the manner described here. However, a value of type `AttributeName` is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.

The `AttributeName` type specifies the name of an XML attribute. A value of type `AttributeName` may be specified using an `AttributeIdentifier`. If the name of the attribute is not specified as a `QualifiedIdentifier`, the `uri` property of the associated QName will be the empty string representing no namespace.

8.3.1 Internal Properties

The `AttributeName` type is logically derived from the Object type and inherits its internal properties. Unless otherwise specified, the `AttributeName` type also inherits the type conversion semantics defined for the Object type (section 9 of ECMA-Script Edition 3). The following table summarises the internal properties the `AttributeName` type adds to those defined by the Object type.

<table>
<thead>
<tr>
<th>Property</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Name]]</code></td>
<td>None</td>
<td>The name of the attribute</td>
</tr>
</tbody>
</table>

The value of the `[[Name]]` property is a value of type QName.

8.4 The `AnyName` Type

The internal `AnyName` type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of E4X must behave as if it produced and operated upon `AnyName` values in the manner described here. However, a value of type `AnyName` is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.

The `AnyName` type is a marker type used to indicate that the name of a particular property was specified as a `WildcardIdentifier` (i.e., "*"). The `AnyName` type has exactly one value, called `anyname`. 
9 Type Conversion

E4X extends the automatic type conversion operators defined in ECMAScript. Unless otherwise specified in this section, the XML type inherits the type conversion semantics defined for the Object type (section 9 of ECMAScript Edition 3).

Note: as in ECMAScript Edition 3, these type conversion functions are internal and are not directly accessible by users. They occur as needed in E4X algorithms and are described here to aid specification of type conversion semantics. In addition, ToString and ToXMLString are exposed indirectly to the E4X user via the built-in methods toString() and toXMLString() defined in sections 12.4.4.37, 12.4.4.38, 12.5.4.19 and 12.5.4.20.

9.1 ToString

E4X extends the behaviour of the ToString operator by specifying its behaviour for the following types.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>Return the XML object as a string as defined in section 9.1.1.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Return the XMLList object as a string as defined in section 9.1.2.</td>
</tr>
<tr>
<td>AttributeName</td>
<td>Given an input argument ( a ), return the result of concatenating the string &quot;@&quot; and ToString(( a._\text{[Name]} )).</td>
</tr>
</tbody>
</table>

9.1.1 ToString Applied to the XML Type

Overview

Given an XML object \( x \), the operator ToString converts \( x \) to a string \( s \). If a value of type XML has simple content (i.e., contains no elements), it represents a primitive value and ToString returns the String contents of the XML object, omitting the start tag, attributes, namespace declarations and end tag. Otherwise, ToString returns an XML encoded string representing the entire XML object, including the start tag, attributes, namespace declarations and the end tag.

Combined with ToString’s treatment of XMLLists (see section 9.1.2), this behaviour allows E4X programmers to access the values of XML leaf nodes in much the same way they access the values of object properties. For example, given a variable named order assigned to the following XML value:

```xml
<order>
    <customer>
        <firstname>John</firstname>
        <lastname>Doe</lastname>
    </customer>
    <item>
        <description>Big Screen Television</description>
        <price>1299.99</price>
        <quantity>1</quantity>
    </item>
</order>
```

the E4X programmer can access individual values of the XML value like this:

```javascript
// Construct the full customer name
var name = order.customer.firstname + " " + order.customer.lastname;

// Calculate the total price
var total = order.item.price * order.item.quantity;
```

E4X does not require the programmer to explicitly select the text nodes associated with each leaf element or explicitly select the first element of each XMLList return value. For cases where this is not the desired behaviour, the ToXMLString operator...
is provided (see section 9.2). Note: in the example above, the String valued properties associated with the XML objects order.item.price and order.item.quantity are implicitly converted to type Number prior to performing the multiply operation.

For XML objects with [[Class]] set to “attribute” or “text”, ToString simply returns their value as a string.

**Semantics**

Given an XML object \( x \), ToString takes the following steps:

1. If \( x.\[[\text{Class}]\] \in \{"attribute", "text"\}, return \( x.\[[\text{Value}]\] \)
2. If \( x.\text{hasSimpleContent()} == \text{true} \)
   a. Let \( s \) be the empty string
   b. For \( i = 0 \) to \( x.\[[\text{Length}]\]-1, \)
      i. If \( x[i].\[[\text{Class}]\] \in \{"comment", "processing-instruction"\} \)
         i. Let \( s \) be the result of concatenating \( s \) and ToString\( (x[i]) \)
   c. Return \( s \)
3. Else
   a. Return ToXMLString\( (x) \)

**9.1.2 ToString Applied to the XMLList Type**

**Overview**

The operator ToString converts an XMLList object \( l \) to a string \( s \). The return value is the string representation of each item in the XMLList concatenated together in order.

Note that the result of calling ToString on a list of size one is identical to the result of calling ToString on the single item contained in the XMLList. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value to simplify the programmer’s task. It allows E4X programmers to access the value of an XMLList containing only a single primitive value in much the same way they access object properties.

**Semantics**

Given an XMLList object \( l \), ToString performs the following steps:

1. If \( l.\text{hasSimpleContent()} == \text{true} \)
   a. Let \( s \) be the empty string
   b. For \( i = 0 \) to \( l.\[[\text{Length}]\]-1, \)
      i. If \( x[i].\[[\text{Class}]\] \notin \{"comment", "processing-instruction"\} \)
         i. Let \( s \) be the result of concatenating \( s \) and ToString\( (l[i]) \)
   c. Return \( s \)
2. Else
   a. Return ToXMLString\( (x) \)

**9.2 ToXMLString ( input argument, [AncestorNamespaces], [IndentLevel] )**

E4X adds the conversion operator ToXMLString to ECMAScript. ToXMLString is a variant of ToString used to convert its argument to an XML encoded string. Unlike ToString, it always includes the start tag, attributes, namespace declarations and end tag associated with an XML element, regardless of content. This is useful in cases where the default ToString behaviour is not desired. The semantics of ToXMLString are specified by the following table.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return <code>ToString(input argument)</code></td>
</tr>
<tr>
<td>Number</td>
<td>Return <code>ToString(input argument)</code></td>
</tr>
<tr>
<td>String</td>
<td>Return <code>EscapeElementValue(input argument)</code></td>
</tr>
<tr>
<td>XML</td>
<td>Create an XML encoded string value based on the content of the XML object</td>
</tr>
</tbody>
</table>
Create an XML encoded string value by calling ToXMLString on each property of the XMLList in order passing the optional argument AncestorNamespaces and concatenating the results to form a single string.

Apply the following steps:
1. Let \( p \) be the result of calling ToPrimitive(input argument, hint String)
2. Let \( s \) be the result of calling ToString\( (p) \)
3. Return EscapeElementValue\( (s) \)

### 9.2.1 ToXMLString Applied to the XML Type

#### Semantics

Given an XML object \( x \) and an optional argument AncestorNamespaces and an optional argument IndentLevel, ToXMLString converts it to an XML encoded string \( s \) by taking the following steps:

1. Let \( s \) be the empty string
2. If IndentLevel was not provided, Let IndentLevel = 0
3. If (XML.prettyPrinting == true)
   a. For \( i = 0 \) to IndentLevel - 1, let \( s \) be the result of concatenating \( s \) and the space <SP> character
4. If \( x[[\text{Class}]] = \text{"text"} \),
   a. If (XML.prettyPrinting == true)
      i. Let \( v \) be the result of removing all the leading and trailing whitespace characters from \( x[[\text{Value}]] \)
      ii. Return EscapeElementValue\( (v) \)
   b. Else
      i. Return EscapeElementValue\( (x[[\text{Value}]])) \)
5. If \( x[[\text{Class}]] = \text{"attribute"}, \) return EscapeAttributeValue\( (x[[\text{Value}]])) \)
6. If \( x[[\text{Class}]] = \text{"comment"}, \) return the result of concatenating the string "<!--", \( x[[\text{Value}]] \) and the string "-->"
7. If \( x[[\text{Class}]] = \text{"processing-instruction"}, \) return the result of concatenating the string "<?", \( x[[\text{Name}}].\text{localName}, \) the space <SP> character, \( x[[\text{Value}]] \) and the string "?>"
8. If AncestorNamespaces was not provided, let AncestorNamespaces = \{ \}
9. Let namespaceDeclarations = \{ \}
10. For each \( ns \) in \( x[[\text{InScopeNamespaces}}] \)
    a. If there is no \( ans \in \{ \text{AncestorNamespaces} \}, \) such that \( ans.\text{uri} = ns.\text{uri} \) and &ns.prefix = ns.prefix
       i. Let \( nsl \) be a copy of \( ns \)
       ii. Let namespaceDeclarations = namespaceDeclarations \( \cup \{ nsl \} \)
       Note: implementations may also exclude unused namespace declarations from namespaceDeclarations
11. Let namespace be the result of calling \([\text{GetNamespace}]\) on \( x[[\text{Name}]] \) with argument (AncestorNamespaces \( \cup \text{namespaceDeclarations} \))
12. If (namespace.prefix == undefined),
    a. Let namespace.prefix be an arbitrary implementation defined namespace prefix, such that there is no \( ns2 \in \) (AncestorNamespaces \( \cup \text{namespaceDeclarations} \)) with namespace.prefix = ns2.prefix
    b. If there is no \( ns2 \in \) (AncestorNamespaces \( \cup \text{namespaceDeclarations} \)), such that \( ns2.\text{uri} = \text{namespace.uri} \) and ns2.prefix = namespace.prefix,
       i. Let namespaceDeclarations = namespaceDeclarations \( \cup \{ \text{namespace} \} \)
13. Let \( s \) be the result of concatenating \( s \) and the string "</".
14. If namespace.prefix is not the empty string,
    a. Let \( s \) be the result of concatenating \( s, \) namespace.prefix and the string ":".
15. Let \( s \) be the result of concatenating \( s \) and \( x[[\text{Name}]].\text{localName} \)
16. Let attrAndNamespaces = \( x[[\text{Attributes}}] \) \( \cup \text{namespaceDeclarations} \)
17. For each \( an \) in attrAndNamespaces
    a. Let \( s \) be the result of concatenating \( s \) and the space <SP> character
    b. If Type\( (an) = \text{XML} \) and \( an[[\text{Class}]] = \text{"attribute"} \)
       i. Let \( ans \) be a copy of the result of calling \([\text{GetNamespace}]\) on \( a[[\text{Name}]] \) with argument AncestorNamespaces
       ii. If (ans.prefix == undefined),
Given a String value quoting conventions in ToXMLString().

Note: implementations may also preserve insignificant whitespace (e.g., inside and between element tags) and attribute quoting conventions in ToXMLString().

Given a String value $s$, the operator EscapeElementValue performs the following steps:

1. Let $r$ be the empty string
2. For each character $c$ in $s$
   a. If ($c$ == ""), let $r$ be the result of concatenating $r$ and the string "&lt;"
   b. Else if ($c$ == ""), let $r$ be the result of concatenating $r$ and the string "&gt;"
   c. Else if ($c$ == ""), let $r$ be the result of concatenating $r$ and the string "&amp;"
   d. Else, let $r$ be the result of concatenating $r$ and $c$
Given a string value \( s \), the operator `EscapeAttributeValue` performs the following steps:

1. Let \( r \) be the empty string
2. For each character \( c \) in \( s \):
   a. If \( (c \text{ is a double quote character (i.e., ")")}) \) let \( r \) be the result of concatenating \( r \) and the string "\&quot;"
   b. Else if \( (c == \"<\") \) let \( r \) be the result of concatenating \( r \) and the string "\&lt;"
   c. Else if \( (c == \"&\") \) let \( r \) be the result of concatenating \( r \) and the string "\&amp;"
   d. Else if \( (c == \u000A) \) let \( r \) be the result of concatenating \( r \) and the string "\&\n"
   e. Else if \( (c == \u000D) \) let \( r \) be the result of concatenating \( r \) and the string "\&\r"
   f. Else let \( r \) be the result of concatenating \( r \) and \( c \)
3. Return \( r \)

9.3 ToXML

E4X adds the operator `ToXML` to ECMAScript. `ToXML` converts its argument to a value of type XML according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Convert the input argument to a string using <code>ToString</code> then convert the result to XML as specified in section 9.3.1.</td>
</tr>
<tr>
<td>Number</td>
<td>Convert the input argument to a string using <code>ToString</code> then convert the result to XML as specified in section 9.3.1.</td>
</tr>
<tr>
<td>String</td>
<td>Create an XML object from the String as specified below in section 9.3.1.</td>
</tr>
<tr>
<td>XML</td>
<td>Return the input argument (no conversion).</td>
</tr>
<tr>
<td>XMLList</td>
<td>If the XMLList contains only one property and the type of that property is XML, return that property. Otherwise, throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>Object</td>
<td>If the <code>[[Class]]</code> property of the input argument is &quot;String&quot;, &quot;Number&quot; or &quot;Boolean&quot;, convert the input argument to a string using <code>ToString</code> then convert the result to XML as specified in section 9.3.1. Otherwise, throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>W3C XML Information Item</td>
<td>Create an XML object from a W3C XML Information Item as specified below in section 9.3.2.</td>
</tr>
</tbody>
</table>

9.3.1 ToXML Applied to the String Type

Overview

When `ToXML` is applied to a string type, it converts it to XML by parsing the string as XML. Prior to conversion, string concatenation can be used to construct portions of the XML object without regard for XML constraints such as well-formedness. For example, consider the following.

```javascript
var John = "<employee><name>John</name><age>25</age></employee>";
var Sue="<employee><name>Sue</name><age>32</age></employee>";
var tagName = "employees";
var employees = new XML("" + tagName + "+" + John + Sue + "+" + tagName + "+"));
```

The last line of this example concatenates several individual strings to form a single string value, then passes the resulting string to the XML constructor function. The XML constructor function uses the internal `ToXML` operator to convert the given string to an XML object.

Semantics

Given a String value \( s \), `ToXML` converts the string to an XML object using the following steps:
1. Let `defaultNamespace` = GetDefaultNamespace()
2. Let `parentString` be the result of concatenating the strings "<parent xmlns='`, `defaultNamespace`, `"'>", `s` and ""</parent>"
3. Parse `parentString` as a W3C Element Information Item `e` and if the parse fails, throw a `SyntaxError` exception
4. Let `x` = `ToXML(e)`
5. If `x`[[`Length`]] == 0
   a. Return a new XML object `t` with `t`[[`Class`]] = "text", `t`[[`Parent`]] = `null` and `t`[[`Value`]] = the empty string
6. else if `x`[[`Length`]] == 1
   a. Let `x[0]`[[`Parent`]] = `null`
   b. Return `x[0]`
7. else throw a `SyntaxError` exception

Note: the use of a W3C XML Information Item is purely illustrative. A W3C XML Information Item is not required to perform this type conversion and implementations may use any mechanism that provides the same semantics.

9.3.2 ToXML Applied to a W3C XML Information Item
Overview

When ToXML is applied to an implementation of a W3C XML Information Item conforming to the W3C XML Information Set specification, it maps the E4X data model onto the given information item such that E4X operators may be used to query, navigate and manipulate the given information item.

The definition of ToXML provided in this section is intentionally general. It permits implementations to create an XML object from a W3C DOM object, XPath node, XQuery item or any other data model that can be mapped to a W3C XML Information Item. E4X does not require implementations to support these data models, nor does it require implementations that support these data models to provide the capability to create XML objects from them. Rather, it defines the semantics of these operations for implementations that choose to implement them.

For example, an implementation may deploy E4X to facilitate manipulating the document object in a web browser. Consequently, an E4X user may write the following E4X code:

```javascript
function createTable() {
  var doc = XML(document); // create an E4X wrapper for the document
  var mytablebody = doc..body.TABLE.TBODY;

  for(j=0;j<2;j++) {
    mytablebody.TR[j] = ""; // append an empty table row
    for(i=0;i<2;i++)        // append a cell with some content
      mytablebody.TR[j].TD[i] = "cell is row " + j + ", column " + i;
  }
  doc..body.TABLE.@border = 2; // set the border attribute of the table
}
```

Instead of writing the equivalent DOM code below:
function createTable () {
    var mybody=document.getElementsByTagName("body").item(0);
    mytable = document.createElement("TABLE");
    mytablebody = document.createElement("TBODY");
    for(j=0;j<2;j++) {
        mycurrent_row=document.createElement("TR");
        for(i=0;i<2;i++) {
            mycurrent_cell=document.createElement("TD");
            currenttext=document.createTextNode("cell is row "+j+", column "+i);
            mycurrent_cell.appendChild(currenttext);
            mycurrent_row.appendChild(mycurrent_cell);
        }
        mytablebody.appendChild(mycurrent_row);
    }
    mytable.appendChild(mytablebody);
    mybody.appendChild(mytable);
    mytable.setAttribute("border","2");
}

Note: the first example above uses the XML constructor function to create an E4X wrapper object around a W3C DOM object (line 2), then performs operations on the resulting E4X object to manipulate the underlying DOM object. Implementations may also provide the capability to use the XML constructor in a new expression (e.g., "var doc = new XML(document);"), to create an independent XML object that does not manipulate the underlying DOM object. As mentioned above, E4X implementations are not required to support the W3C DOM or to provide the capability to construct XML objects from W3C DOM objects.

Semantics

A W3C Information Item i is mapped onto an XML type x as follows:

1. Map x.[[Parent]] to null
2. If i is a character information item
   a. Map x.[[Class]] to "text"
   b. Map x.[[Value]] to the largest contiguous sequence of character information items that have the same parent starting with i and continuing forward in document order. Map each character of the string x.[[Value]] to the corresponding [character code] property of each character information item in the sequence.
   c. If (XML.ignoreWhitespace == true) and (each character in x.[[Value]] is a XMLWhitespace), return null
   d. Else return x
3. If i is a comment information item
   a. If XML.ignoreComments == true, return null
   b. Map x.[[Class]] to "comment"
   c. Map x.[[Value]] to the [content] property of i
   d. Return x
4. If i is a processing instruction information item
   a. If XML.ignoreProcessingInstructions == true, return null
   b. Map x.[[Class]] to "processing-instruction"
   c. Map x.[[Name]] to the [target] property of i
   d. Map x.[[Value]] to the [content] property of i
   e. Return x
5. If i is an attribute information item
   a. Map x.[[Class]] to "attribute"
   b. Map x.[[Name]].localName to the [local name] property of i
   c. Map x.[[Name]].uri to the [namespace name] property of i
   Note: implementations may also map x.[[Name]].[[Prefix]] to the [prefix] property of i.
d. Map \(x.\{\text{Value}\}\) to the \textbf{normalized value} property of \(i\)

6. If \(i\) is an element information item
   a. Map \(x.\{\text{Class}\}\) to the \textbf{element} value
   b. Map \(x.\{\text{Name}\}.\text{localName}\) to the \textbf{local name} property of \(i\)
   c. Map \(x.\{\text{Name}\}.\text{uri}\) to the \textbf{namespace name} property of \(i\)
      Note: implementations may also map \(x.\{\text{Name}\}.\{\text{Prefix}\}\) to the \textbf{prefix} property of \(i\).
   d. For each attribute information item \(a\) in the \textbf{attributes} property of \(i\)
      i. Map a member \(a.\{\text{attr}\}\) of \(x.\{\text{Attributes}\}\) to the result of calling ToXML\((a)\)
      ii. Map \(a.\{\text{Parent}\}\) to \(x\)
   
   e. Note: The E4X data model does not enforce the constraint: \(\forall x \in \text{XML}: x.\{\text{InScopeNamespaces}\} \supseteq x.\{\text{Parent}\}.\{\text{InScopeNamespaces}\}\). However, implementations may at this point map members of \(x.\{\text{Parent}\}.\{\text{InScopeNamespaces}\}\) to members of \(x.\{\text{InScopeNamespaces}\}\).
   f. For each attribute information item \(a\) in the \textbf{namespace attributes} property of \(i\)
      i. Map a member \(a.\{\text{ns}\}\) of \(x.\{\text{InScopeNamespaces}\}\) to \(a\) as follows:
         1. if the \textbf{local name} property of \(a\) is "xmlns"
            a. Map \(a.\{\text{prefix}\}\) to the empty string
         2. else
            a. Map \(a.\{\text{prefix}\}\) to the \textbf{local name} property of \(a\)
            b. Map \(a.\{\text{uri}\}\) to the \textbf{normalized value} property of \(a\)
   
   g. Let \(j = 0\)
   h. Let \(xmlChild = 0\)
   i. Let \(numItemChildren\) be the number of information items in the \textbf{children} property of \(i\)
   j. While (\(j < numItemChildren\))
      i. Let \(item\) be the \(j\)th information item in the \textbf{children} property of \(i\)
      ii. Let \(c = \text{ToXML}(item)\)
      iii. If \(c\) is not \textbf{null}
         1. Map \(x.\{\text{xmlChild}\}\) to \(c\)
         2. Map \(x.\{\text{xmlChild}\}.\{\text{Parent}\}\) to \(x\)
         3. if \(c.\{\text{Class}\}\) == "text"
            a. Let \(j = j + c.\{\text{Value}\}.\text{length} - 1\)
         4. Let \(xmlChild = xmlChild + 1\)
      iv. Let \(j = j + 1\)
   k. Map \(x.\{\text{Length}\}\) to \(xmlChild\)
   l. Return \(x\)

7. if \(i\) is a document information item
   a. Return the result of calling ToXML on the \textbf{document element} property of \(i\)

8. if \(i\) is an unexpanded entity reference information item
   a. Throw a \textbf{TypeError} exception

9. Return \textbf{null}
   Note: ToXML ignores document type declaration information items

\section*{9.4 ToXMLList}

E4X adds the operator ToXMLList to ECMAScript. ToXMLList converts its argument to a value of type XMLList according to the following table:

<table>
<thead>
<tr>
<th>\textbf{Input Type}</th>
<th>\textbf{Result}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a \textbf{TypeError} exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a \textbf{TypeError} exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Convert the input argument to a string using ToString then convert the result to XMLList as specified in section 9.4.1.</td>
</tr>
<tr>
<td>Number</td>
<td>Convert the input argument to a string using ToString then convert the result to XMLList as specified in section 9.4.1.</td>
</tr>
<tr>
<td>String</td>
<td>Create an XMLList object from the String as specified below in section 9.4.1.</td>
</tr>
<tr>
<td>XML</td>
<td>Given an input argument (x), return a new XMLList object (l) with (l.{\text{Length}} = 1), (l[0] = x), (l.{\text{TargetObject}} = x.{\text{Parent}}) and (l.{\text{Parent}}).</td>
</tr>
</tbody>
</table>
9.4.1 ToXMLList Applied to the String Type

Overview

When ToXMLList is applied to a string type, it converts the string type to an XMLList by parsing the string as an XML fragment. Prior to conversion, string concatenation can be used to construct portions of the XMLList value. For example,

```javascript
var John = "<employee><name>John</name><age>25</age></employee>";
var Sue = "<employee><name>Sue</name><age>32</age></employee>";
var l = new XMLList(John + Sue);
```

The last line of this example concatenates two strings to form a single string value, then passes the resulting string to the XMLList constructor function. The XML constructor function uses the internal ToXMLList operator to convert the given string to an XML object.

Semantics

Given a String value \( s \), ToXMLList converts it to an XMLList using the following steps:

1. Let defaultNamespace = GetDefaultNamespace()
2. Let parentString be the result of concatenating the strings "<parent xmlns='", defaultNamespace, "'>", \( s \) and "</parent>";
3. Parse parentString as a W3C Element Information Item \( e \)
4. If the parse fails, throw a SyntaxError exception
5. Let \( x = \) ToXML(\( e \))
6. Let \( l \) be a new XMLList with \( l.[[TargetObject]] = \) null
7. For \( i = 0 \) to \( x.[[Length]]-1 \)
   a. Let \( x[i].[[Parent]] = \) null
   b. Call the [[Append]] method of \( l \) with argument \( x[i] \)
8. Return \( l \)

Note: the use of a W3C XML Information Item is purely illustrative. A W3C XML Information Item is not required to perform this type conversion and implementations may use any mechanism that provides the same semantics.

9.5 ToAttributeName

E4X adds the operator ToAttributeName to ECMAScript. ToAttributeName converts its argument to a value of type AttributeName according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Number</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>String</td>
<td>Create an AttributeName from the String as specified below in section 9.5.1</td>
</tr>
<tr>
<td>XML</td>
<td>Convert the input argument to a string using ToString then convert the result to an AttributeName as specified in section 9.5.1.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Convert the input argument to a string using ToString then convert the result to an AttributeName as specified in section 9.5.1.</td>
</tr>
</tbody>
</table>
| Object     | If the input argument is a QName object (i.e., its internal \([[[Class]]]\) property is
"QName"), return a new AttributeName with its [[Name]] property set to the input argument. Otherwise, convert the input argument to a string using ToString then convert the result to an AttributeName as specified in section 9.5.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeName</td>
<td>Return the input argument (no conversion).</td>
</tr>
<tr>
<td>AnyName</td>
<td>Return the result of calling ToAttributeName(&quot;*&quot;)</td>
</tr>
</tbody>
</table>

9.5.1 ToAttributeName Applied to the String Type

Given a string s, the ToAttributeName conversion function returns an AttributeName a. The [[Name]] property of a is set to a new QName q with its local name set to the given string and its URI set to the empty string representing no namespace.

Semantics

Given a String value s, ToAttributeName converts it to an AttributeName using the following steps:

1. Let ns be a new Namespace created as if by calling the constructor new Namespace()
2. Let q be a new QName created as if by calling the constructor new QName(ns, s)
3. Return a new AttributeName a with a.[[Name]] = q

9.6 ToXMLName

E4X adds the operator ToXMLName to ECMAScript. ToXMLName is an internal operator that converts its argument to a value of type AttributeName or a QName object according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Number</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>String</td>
<td>Create a QName object or AttributeName from the String as specified below in section 9.6.1.</td>
</tr>
<tr>
<td>XML</td>
<td>Convert the input argument to a string using ToString then convert the result to a QName object or AttributeName as specified in section 9.6.1.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Convert the input argument to a string using ToString then convert the result to a QName object or AttributeName as specified in section 9.6.1.</td>
</tr>
<tr>
<td>Object</td>
<td>If the input argument is a QName object (i.e., its [[Class]] property is &quot;QName&quot;), return the input argument. Otherwise, convert the input argument to a string using ToString, then convert the result to a QName object or AttributeName as specified in section 9.6.1.</td>
</tr>
<tr>
<td>AttributeName</td>
<td>Return the input argument (no conversion).</td>
</tr>
<tr>
<td>AnyName</td>
<td>Return the result of calling ToXMLName(&quot;*&quot;)</td>
</tr>
</tbody>
</table>

9.6.1 ToXMLName Applied to the String Type

Given a string s, the ToXMLName conversion function returns a QName object or AttributeName. If the first character of s is "@", ToXMLName creates an AttributeName using the ToAttributeName operator. Otherwise, it creates a QName object using the QName constructor.

Semantics

Given a String value s, ToXMLName operator converts it to a QName object or AttributeName using the following steps:

1. If the first character of s is "@
   a. Let name = s.substring(1, s.length)
   b. Return ToAttributeName(name)
2. Else
a. Return a QName object created as if by calling the constructor new QName(s)
10 Expressions

10.1 Primary Expressions

Syntax

E4X extends the primary expressions defined by ECMAScript with the following production:

\[
\text{PrimaryExpression} : \\
\text{PropertyIdentifier} \\
\text{XMLInitialiser} \\
\text{XMLListInitialiser}
\]

\[
\text{PropertyIdentifier} : \\
\text{AttributeIdentifier} \\
\text{QualifiedIdentifier} \\
\text{WildcardIdentifier}
\]

Semantics

The production \text{PrimaryExpression : PropertyIdentifier} is evaluated as follows:

1. Let \(n\) be the result of evaluating \text{PropertyIdentifier}
2. Let \(name = \text{ToXMLName}(n)\)
3. While (true)
   a. If there are no more objects on the scope chain,
      i. return \text{undefined}
   b. Let \(o\) be the next object on the scope chain.
      Note: on the first iteration, \(o\) will be the first object on the scope chain
   c. If \(\text{Type}(o) \in \{\text{XML}, \text{XMLList}\}\)
      i. Let \(\text{hasProp}\) be the result of calling the [[HasProperty]] method of \(o\), passing \(name\) as the property
      ii. If \(\text{hasProp} = \text{true}\)
          1. Return a value of type Reference whose base object is \(o\) and whose property name is \(name\)

The production \text{PropertyIdentifier : AttributeIdentifier} is evaluated as follows:

1. Return the result of evaluating \text{AttributeIdentifier}

The productions \text{PropertyIdentifier : QualifiedIdentifier}, and \text{PropertyIdentifier : WildcardIdentifier} are evaluated exactly the same manner except \text{AttributeIdentifier} is replaced by \text{QualifiedIdentifier} and \text{WildcardIdentifier} in step 1.

10.1.1 Attribute Identifiers

Syntax

E4X extends ECMAScript by adding attribute identifiers. The syntax of an attribute identifier is specified by the following production:

\[
\text{AttributeIdentifier} : \\
\@ \text{PropertySelector} \\
\@ \text{QualifiedIdentifier} \\
\@ [ \text{Expression} ]
\]

\[
\text{PropertySelector} : \\
\text{Identifier} \\
\text{WildcardIdentifier}
\]
Overview

An AttributeIdentifier is used to identify the name of an XML attribute. It evaluates to a value of type AttributeName. The preceding “@” character distinguishes a XML attribute from a XML element with the same name. This AttributeIdentifier syntax was chosen for consistency with the familiar XPath syntax.

Semantics

The production AttributeIdentifier : @ PropertySelector is evaluated as follows:

1. Let name be a string value containing the same sequence of characters as in the PropertySelector
2. Return ToAttributeName(name)

The production AttributeIdentifier : @ QualifiedIdentifier is evaluated as follows:

1. Let q be the result of evaluating QualifiedIdentifier
2. Return ToAttributeName(q)

The production AttributeIdentifier : @ [ Expression ] is evaluated as follows:

1. Let e be the result of evaluating Expression
2. Return ToAttributeName(GetValue(e))

10.1.2 Qualified Identifiers

Syntax

E4X extends ECMAScript by adding qualified identifiers. The syntax for qualified identifiers is specified by the following productions:

QualifiedIdentifier :
    PropertySelector :: PropertySelector
    PropertySelector :: [ Expression ]

Overview

QualifiedIdentifiers are used to identify values defined within a specific namespace. They may be used to access, manipulate and create namespace qualified XML element and attribute names. For example,
// Create a SOAP message
var message = <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap:Body>
      <m:GetLastTradePrice xmlns:m="http://mycompany.com/stocks">
        <symbol>DIS</symbol>
      </m:GetLastTradePrice>
    </soap:Body>
</soap:Envelope>

// declare the SOAP and stocks namespaces
var soap = new Namespace("http://schemas.xmlsoap.org/soap/envelope/");
var stock = new Namespace("http://mycompany.com/stocks");

// extract the soap encoding style and body from the soap message
var encodingStyle = message.@soap::encodingStyle;
var body = message.soap::Body;

// change the stock symbol
message.soap::Body.stock::GetLastTradePrice.symbol = "MYCO";

Semantics

A QualifiedIdentifier evaluates to a QName object. The production QualifiedIdentifier : PropertySelector :: PropertySelector is evaluated as follows:

1. Let ns be the result of evaluating the first PropertySelector
2. If (ns == anyname), let ns = null
3. Let localName be a string value containing the same sequence of characters as in the second PropertySelector
4. Return a new QName created as if by calling the constructor new QName(GetValue(ns), localName)

The production QualifiedIdentifier : PropertySelector :: [ Expression ] is evaluated as follows:

1. Let ns be the result of evaluating PropertySelector
2. If (ns == anyname), let ns = null
3. Let e be the result of evaluating Expression
4. Return a new QName created as if by calling the constructor new QName(GetValue(ns), GetValue(e))

10.1.3 Wildcard Identifiers

Syntax

E4X extends ECMAScript by adding a wildcard identifier. The syntax of the wildcard identifier is specified by the following production:

   WildcardIdentifier :
       *

Overview

The WildcardIdentifier is used to identify any name. It may be used for matching namespaces, properties of XML objects or XML attributes. The wildcard identifier evaluates to the value anyname.

Semantics
The production `WildcardIdentifier : *` is evaluated as follows:

1. Return `anyname`

### 10.1.4 XML Initialiser

#### Overview

An XML initialiser is an expression describing the initialization of an XML object, written in a form of a literal. It may specify an XML element, an XML comment, an XML PI, or a CDATA section using ordinary XML syntax. For XML elements, it provides the name, XML attributes and XML properties of an XML object.

The syntactic grammar for XML initialisers is used to find the end of XML initialisers and evaluate embedded expressions. It permits sequences of characters that are not valid XML syntax. After evaluating all embedded expressions, the resulting string of characters comprising the XML value is passed to the XML constructor, which parses it according to the more stringent XML grammar.

The syntactic grammar for XML initialisers processes input elements produced by the lexical grammar goal symbols `InputElementXMLTag` and `InputElementXMLContent`. These input elements are described in section 7.3.

Below are some examples of XML initialisers.

```javascript
// an XML object representing a person with a name and age
var person = <person><name>John</name><age>25</age></person>;

// a variable containing an XML object representing two employees
var e = <employees>
    <employee id="1"><name>Joe</name><age>20</age></employee>
    <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;
```

Expressions may be used to compute parts of an XML initialiser. Expressions are delimited by curly braces and may appear inside tags or element content. Inside a tag, expressions may be used to compute a tag name, attribute name, or attribute value. Inside an element, expressions may be used to compute element content. For example,

```javascript
for (i = 0; i < 10; i++)
    e[i] = <employee id={i}>    // compute id value
        <name>{names[i].toUpperCase()}</name> // compute name value
        <age>{ages[i]}</age>    // compute age value
    </employee>;
```

Each expression is evaluated and replaced by its value prior to parsing the literal XML value. For example the following expression,

```javascript
var tagnname = "name";
var attributename = "id";
var attributevalue = 5;
var content = "Fred";

var x = <{tagnname} {attributename}={attributevalue}>{content}</{tagnname}>;
```

would assign the following XML value to the variable `x`. 

---
Syntax

XMLInitialiser :
    XMLMarkup
    XMLElement

XMLElement :
    < XMLTagContent />
    < XMLTagContent > XMLElementContent_opt </ XMLTagContent >

XMLTagContent :
    XMLTagCharacters XMLTagContent_opt
    XMLWhitespace XMLTagContent_opt
    { Expression } XMLTagContent_opt
    = XMLWhitespace_opt { Expression } XMLTagContent_opt
    = XMLWhitespace_opt XMLAttributeValue XMLTagContent_opt

XMLElementContent :
    XMLMarkup XMLElementContent_opt
    XMLText XMLElementContent_opt
    XMLEl ement XMLElementContent_opt
    { Expression } XMLElementContent_opt

Semantics

The production XMLInitialiser : XMLMarkup is evaluated as follows:

1. Let markup be a string literal containing the same sequence of characters as XMLMarkup
2. Return a new XML object created as if by calling the XML constructor with argument markup (section 12.4.2)

The production XMLInitialiser : XMLElement is evaluated as follows:

3. Let element be a the result of evaluating XMLElement
4. Return a new XML object created as if by calling the XML constructor with argument element (section 12.4.2)

The production XMLElement : < XMLTagContent /> is evaluated as follows:

1. Let content be the result of evaluating XMLTagContent
2. Return the result of concatenating the string value "<", followed by content, followed by the string value "/>

The production XMLElement : < XMLTagContent > XMLElementContent_opt </ XMLTagContent > is evaluated as follows:

1. Let startTag be the result of evaluating the first XMLTagContent
2. Let content be the result of evaluating XMLElementContent; if not present, use the empty string
3. Let endTag be the result of evaluating the second XMLTagContent
4. Return the result of concatenating the string value "<", followed by startTag, followed by the string value "/>
   followed by content, followed by the string value "<!", followed by endTag, followed by the string value ">[]"
1. Let whitespace be a string literal containing the string " " (space)
   Note: implementations may also preserve insignificant whitespace inside XML tags by setting whitespace to the same sequence of characters as XMLWhitespace
2. Let tagContent be the result of evaluating XMLTagContent; if not present, use the empty string
3. Return the result of concatenating whitespace followed by tagContent

The production XMLTagContent : { Expression } XMLTagContent_opt is evaluated as follows:

1. Let expRef be the result of evaluating Expression
2. Let expression = GetValue(expRef)
3. Let tagContent be the result of evaluating XMLTagContent; if not present, use the empty string
4. Return the result of concatenating ToString(expression), followed by tagContent

The production XMLTagContent : = XMLWhitespace_opt { Expression } XMLTagContent_opt is evaluated as follows:

1. Let attributeValue be a string literal containing the same sequence of characters as XMLAttributeValue
2. Let tagContent be the result of evaluating XMLTagContent; if not present, use the empty string
3. Let whitespace be the empty string
   Note: implementations may also preserve insignificant whitespace inside XML tags by setting whitespace to the same sequence of characters as XMLWhitespace
4. Return the result of concatenating equals "=" followed by whitespace, followed by attributeValue, followed by tagContent

The production XMLElementContent : XMLMarkup XMLElementContent_opt is evaluated as follows:

1. Let markup be the result of evaluating XMLMarkup
2. Let content be the result of evaluating XMLElementContent; if not present, use the empty string
3. Return the result of concatenating markup followed by content

The production XMLElementContent : XMLElement XMLElementContent_opt is evaluated as follows:

1. Let element be the result of evaluating XMLMarkup
2. Let content be the result of evaluating XMLElementContent; if not present, use the empty string
3. Return the result of concatenating element followed by content

The production XMLElementContent : XMLText XMLElementContent_opt is evaluated as follows:

1. Let text be a string literal containing the same sequence of characters as XMLText
2. Let content be the result of evaluating XMLElementContent; if not present, use the empty string
3. Return the result of concatenating text, followed content.

The production XMLElementContent : { Expression } XMLElementContent_opt is evaluated as follows:

1. Let expRef be the result of evaluating Expression
2. Let expression = GetValue(expRef)
3. If Type(expression) ∈ {XML, XMLList},
   a. Let value be the result of calling ToXMLString(expression)
4. Else
a. Let value be the result of calling EscapeElementValue(ToString(expression))
5. Let content be the result of evaluating XMLElementContent; if not present, use the empty string
6. Return the result of concatenating value followed by content

10.1.5 XMLList Initialiser

Overview

An XMLList initialiser is an expression describing the initialization of an XMLList object written in a form resembling a literal. It describes an ordered list of XML properties using an anonymous XML element syntax. XMLList initialisers begin with the character sequence “<>” and end with the character sequence “</>”.

The syntactic grammar for XML initialisers processes input elements produced by the lexical grammar goal symbols InputElementXMLTag and InputElementXMLContent. These input elements are described in section 7.3.

Below are some examples of XMLList Initialisers,

```javascript
var docfrag = <> <name>Phil</name> <age>35</age> <hobby>skiing</hobby> </;
var emplist = <>
  <employee id="0" ><name>Jim</name> <age>25</age></employee>
  <employee id="1" ><name>Joe</name> <age>20</age></employee>
  <employee id="2" ><name>Sue</name> <age>30</age></employee>
</;
```

Syntax

XMLListInitialiser:

```javascript
< > XMLElementContent </ >
```

Semantics

The production XMLList : < > XMLElementContent_opt </ > is evaluated as follows:

1. Let content be the result of evaluating XMLElementContent; if not specified use the empty string
2. Return a new XMLList object created as if by calling the XMLList constructor with argument content

10.2 Left-Hand-Side Expressions

E4X extends the left-hand-side expressions defined in ECMAScript with the following productions:

```javascript
MemberExpression :
  MemberExpression . PropertyIdentifier
  MemberExpression .. Identifier
  MemberExpression . ( Expression )
```

```javascript
CallExpression :
  CallExpression . PropertyIdentifier
  CallExpression .. Identifier
  CallExpression .. PropertyIdentifier
  CallExpression . ( Expression )
```

In addition, E4X defines new semantics for existing left-hand-side expressions applied to values of type XML and XMLList.

10.2.1 Property Accessors

Syntax
E4X reuses and extends ECMAScript’s property accessor syntax for accessing properties and XML attributes within values of type XML and XMLList. XML properties may be accessed by name, using either the dot notation:

```
MemberExpression . Identifier
MemberExpression . PropertyIdentifier
CallExpression . Identifier
CallExpression . PropertyIdentifier
```

or the bracket notation:

```
MemberExpression [ Expression ]
CallExpression [ Expression ]
```

**Overview**

When `MemberExpression` or `CallExpression` evaluate to a XML object, the property accessor uses the XML `[[Get]]` method to determine the result. If the right operand evaluates to an array index, the XML `[[Get]]` method converts the XML object to an XMLList and retrieves the requested property from the resulting XMLList. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value. Otherwise, the XML `[[Get]]` method examines the XML properties and XML attributes of the left operand and returns an XMLList containing the ones with names that match its right operand in order. For example,

```html
var order = <order id="123456" timestamp="Mon Mar 10 2003 16:03:25 GMT-0800 (PST)">
  <customer>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
  </customer>
  <item>
    <description>Big Screen Television</description>
    <price>1299.99</price>
    <quantity>1</quantity>
  </item>
</order>;
```

```javascript
var customer = order.customer; // get the customer element from the order
var id = order.@id;       // get the id attribute from the order
var orderChildren = order.*; // get all the child elements from the order element
var orderAttributes = order.@*; // get all the attributes from the order element
var firstOrder = order[0];  // get the first (and only) order [treating single element as a list]
```

When `MemberExpression` or `CallExpression` evaluate to an XMLList, the property accessor uses the XMLList `[[Get]]` method to determine the result. If the bracket notation is used with a numeric identifier, the XMLList `[[Get]]` method simply returns the property of the left operand with a property name matching the numeric identifier. Otherwise, the XMLList `[[Get]]` method applies the property accessor operation to each XML object in the list and returns a new XMLList containing the results in order. For example,
var order = `<order> 
  <customer> 
    <firstname>John</firstname> 
    <lastname>Doe</lastname> 
  </customer> 
  <item id = "3456"> 
    <description>Big Screen Television</description> 
    <price>1299.99</price> 
    <quantity>1</quantity> 
  </item> 
  <item id = "56789"> 
    <description>DVD Player</description> 
    <price>399.99</price> 
    <quantity>1</quantity> 
  </item> 
</order>;

var descriptions = order.item.description;  // get the list of all item descriptions 
var itemIds = order.item.@id;          // get the list of all item id attributes 
var secondItem = order.item[1];        // get second item by numeric index 
var itemChildren = order.item.*;       // get the list of all child elements in all item elements 
var secondChild = order.item.*[1];     // get the second child element from the order by index 

In the first property accessor statement above, the expression “order.item” examines the XML properties of the XML object bound to “order” and returns an XMLList containing the two XML objects named “item”. The expression “order.item.description” then examines the XML properties of each item in the resulting XMLList and returns an XMLList containing the two XML objects named “description”.

When `MemberExpression` or `CallExpression` do not evaluate to a value of type XML or XMLList and the right hand side of the expression is an `Identifier`, the property accessor performs according to the semantics specified in ECMAScript Edition 3. However, if `MemberExpression` or `CallExpression` do not evaluate to a value of type XML or XMLList and the right hand side of the expression is a `PropertyIdentifier`, the property accessor throws a `TypeError` exception.

**Semantics**

As in ECMAScript Edition 3, the behaviour of the production:

```
MemberExpression : MemberExpression . Identifier
```

is identical to the behaviour of the production:

```
MemberExpression : MemberExpression [ <identifier-string> ]
```

and similarly, the behaviour of the production:

```
CallExpression : CallExpression . Identifier
```

is identical to the behaviour of the production:

```
CallExpression : CallExpression [ <identifier-string> ]
```

where `<identifier-string>` is a string literal containing the same sequence of characters as the `Identifier`. 
The production \textit{MemberExpression} : \textit{MemberExpression} \[ \textit{Expression} \] is evaluated as follows:

1. Let \( oRef \) be the result of evaluating \textit{MemberExpression}
2. Let \( o = \text{ToObject}(\text{GetValue}(oRef)) \)
3. Let \( pRef \) be the result of evaluating \textit{Expression}
4. Let \( p = \text{GetValue}(pRef) \)
5. If \( \text{Type}(o) \in \{ \text{XML, XMLList} \} \)
   a. Return a value of type Reference whose base object is \( o \) and whose property name is \( \text{ToXMLName}(p) \)
6. Else
   a. If \( \text{Type}(p) \in \{ \text{AttributeName, AnyName} \} \) or \( \text{Type}(p) \) is Object and \( p.\text{[Class]} = \text{"QName"} \)
      i. Throw a \text{TypeError} exception
   b. Return a value of type Reference whose base object is \( o \) and whose property name is \( \text{ToString}(p) \)

The production \textit{CallExpression} : \textit{CallExpression} \[ \textit{Expression} \] is evaluated in exactly the same manner, except the contained \textit{CallExpression} is evaluated in step 1.

The production \textit{MemberExpression} : \textit{MemberExpression} . \textit{PropertyIdentifier} behaves exactly as the production \textit{MemberExpression} : \textit{MemberExpression} \[ \textit{Expression} \], except the contained \textit{PropertyIdentifier} is evaluated in step 3. Similarly, the production \textit{CallExpression} : \textit{CallExpression} . \textit{PropertyIdentifier} behaves exactly as the production \textit{CallExpression} : \textit{CallExpression} \[ \textit{Expression} \], except the contained \textit{PropertyIdentifier} is evaluated in step 3.

\subsection{10.2.2 Function Calls}

\textbf{Syntax}

E4X reuses ECMAScript’s function call syntax for invoking methods on values of type XML and XMLList. The ECMAScript syntax for function calls is described by the following productions:

\begin{verbatim}
CallExpression
  MemberExpression Arguments

Arguments :
  ()
  ( ArgumentList )

ArgumentList :
  AssignmentExpression
  ArgumentList , AssignmentExpression
\end{verbatim}

\textbf{Overview}

Unlike values of type Object, values of type XML and XMLList store and retrieve properties separately from methods so that XML method names do not clash with XML property names. For example,

\begin{verbatim}
var rectangle = <rectangle>
  <x>50</x>
  <y>75</y>
  <length>20</length>
  <width>30</width>
</rectangle>;

var numChildren = rectangle.length();     // returns 4 – number of children in <rectangle>
var rectangleLength = rectangle.length;   // returns 20 – content of <length> element
rectangle.length = 50;                    // change the length element of the rectangle
\end{verbatim}
To accomplish this, E4X modifies the semantics of the call expression to invoke the operation CallMethod (see section 10.2.2.1).

**Semantics**

The production `CallExpression : MemberExpression Arguments` is evaluated as follows:

1. Let `r` be the result of evaluating `MemberExpression`
2. Let `args` be the result of evaluating `Arguments`, producing an internal list of argument values
3. Return the result of calling the operation `CallMethod(r, args)`

**10.2.2.1 CallMethod ( r , args )**

**Overview**

The CallMethod abstract operator is used in this specification for invoking functions and methods. This operator is not part of the language. It is defined here to aid the specification of the language.

When the operation CallMethod is called with a single parameter `r`, it first checks to see if `r` is a Reference. If it is not, it attempts to call `r` as a function. However, if `r` is a Reference, it extracts the base and property name from the Reference `r`. Then, CallMethod calls the Object `[[Get]]` method to retrieve the property of the base object with the given property name.

Note: The XML and XMLList `[[Get]]` method is never called for method lookup.

If no such property exists and `base` is an XMLList of size 1, CallMethod delegates the method invocation to the single property it contains. This treatment intentionally blurs the distinction between XML objects and XMLLists of size 1.

If no such property exists and `base` is an XML object containing no XML valued children (i.e., an attribute, leaf node or element with simple content), CallMethod attempts to delegate the method lookup to the string value contained in the leaf node. This treatment allows users to perform operations directly on the value of a leaf node without having to explicitly select it. For example,

```plaintext
var shipto= <shipto>
    <name>Fred Jones</name>
    <street>123 Foobar Ave.</street>
    <citystatezip>Redmond, WA, 98008</citystatezip>
</shipto>
```

```plaintext
var upperName = shipto.name.toUpperCase(); // calls String.toUpperCase on value of text node
var citystatezip = shipto.citystatezip.split(" "); // calls String.split() on text node to parse address
var state = citystatezip[1]; // line into individual city, state and zip values
var zip = citystatezip[2];
```

**Semantics**

Given a Reference `r` and a list of arguments `args`, the operation CallMethod performs the following steps:

1. Let `f = r`
2. Let `base = null`
3. If `Type(r)` is Reference
   a. Let `base = GetBase(r)`
   b. If `base === null`, throw a `ReferenceException`
   c. Let `P = GetPropertyName(r)`
   d. Let `f` be the result of calling the Object `[[Get]]` method with `base` as the this object and argument `P`
   e. If `f === undefined` and `Type(base)` is XMLList and `base[[Length]] === 1`
      i. Let `r0` be a new Reference with `base object = base[0]` and `property name = P`
      ii. Return the result of calling CallMethod(r0, args) recursively
f. If \( f == \text{undefined} \) and Type(base) is XML and base.hasSimpleContent() == true
   i. Let \( r_0 \) be a new Reference with base object = ToObject(ToString(base)) and property name = \( P \)
   ii. Return the result of calling CallMethod(\( r_0, \text{args} \)) recursively

4. If Type(f) is not Object, throw a TypeError exception
5. If \( f \) does not implement the internal \([\text{Call}]\) method, throw a TypeError exception
6. If base is an activation object, base = null
7. Return the result of calling the \([\text{Call}]\) method of \( f \) providing base as the this value and the list \( \text{args} \) as the argument values

10.2.3 XML Descendant Accessor

Syntax

E4X extends ECMAScript by adding a descendant accessor. The following productions describe the syntax of the descendant accessor:

\[
\text{MemberExpression} : \quad \text{MemberExpression} \ldots \text{Identifier} \\
\text{MemberExpression} : \quad \text{MemberExpression} \ldots \text{PropertyIdentifier}
\]

\[
\text{CallExpression} : \quad \text{CallExpression} \ldots \text{Identifier} \\
\text{CallExpression} : \quad \text{CallExpression} \ldots \text{PropertyIdentifier}
\]

Overview

When the MemberExpression or CallExpression evaluate to an XML object or an XMLList, the descendant accessor examines all of the descendant XML properties (i.e., children, grand children, great-grandchildren, etc) of its left operand and returns an XMLList containing those with names that match its right operand in order. For example,

```javascript
var e = <employees>
    <employee id="1"><name>Joe</name><age>20</age></employee>
    <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;
var names = e..name; // get all the names in e
```

Semantics

The production MemberExpression : MemberExpression .. Identifier is evaluated as follows:

1. Let \( \text{ref} \) be the result of evaluating MemberExpression
2. Let \( x = \text{GetValue(ref)} \)
3. If Type(\( x \)) \( \notin \) \{XML, XMLList\}, throw a TypeError exception
4. Let \( P \) be a string value containing the same sequence of characters as Identifier
5. Return the result of calling the \([\text{Descendants}]\) method of \( x \) with argument \( P \)

The production CallExpression : CallExpression .. Identifier is evaluated in exactly the same manner, except that the contained CallExpression is evaluated in step 1.

The production MemberExpression : MemberExpression .. PropertyIdentifier is evaluated as follows:

1. Let \( \text{ref} \) be the result of evaluating MemberExpression
2. Let \( x = \text{GetValue(ref)} \)
3. If Type(\( x \)) \( \notin \) \{XML, XMLList\}, throw a TypeError exception
4. Let \( P \) be the result of evaluating PropertyIdentifier
5. Return the result of calling the \([\text{Descendants}]\) method of \( x \) with argument \( P \)
The production $CallExpression : CallExpression \ldots PropertyIdentifier$ is evaluated in exactly the same manner, except that the contained $CallExpression$ is evaluated in step 1.

### 10.2.4 XML Filtering Predicate Operator

**Syntax**

E4X extends ECMAScript by adding a filtering predicate operator. The following productions describe the syntax of the filtering predicate operator:

- MemberExpression:
  - MemberExpression . ( Expression )

- CallExpression:
  - CallExpression . ( Expression )

**Overview**

When the left operand evaluates to an XML object, the filtering predicate adds the left operand to the front of the scope chain of the current execution context, evaluates the $Expression$ with the augmented scope chain, converts the result to a Boolean value, then restores the scope chain. If the result is true, the filtering predicate returns an XMLList containing the left operand. Otherwise it returns an empty XMLList.

When the left operand is an XMLList, the filtering predicate is applied to each XML object in the XMLList in order using the XML object as the left operand and the $Expression$ as the right operand. It concatenates the results and returns them as a single XMLList containing all the XML properties for which the result was true. For example,

```javascript
var john = e.employee.(name == "John");  // employees with name John
var twoemployees = e.employee.(@id == 0 || @id == 1);  // employees with id's 0 & 1
var emp = e.employee.(@id == 1).name;  // name of employee with id 1
```

The effect of the filtering predicate is similar to SQL’s WHERE clause or XPath’s filtering predicates.

For example, the statement:

```javascript
// get the two employees with ids 0 and 1 using a predicate
var twoEmployees = e.employee.(@id == 0 || @id == 1);
```

produces the same result as the following set of statements:

```javascript
// get the two employees with the ids 0 and 1 using a for loop
var i = 0;
var twoEmployees = new XMLList();
for (var p in e.employee) {
    with (p) {
        if (@id == 0 || @id == 1) {
            twoEmployees[i++] = p;
        }
    }
}
```

**Semantics**

The production $MemberExpression : MemberExpression . ( Expression )$ is evaluated as follows::
1. Let objref be the result of evaluating MemberExpression
2. Let \( x = \text{GetValue}(\text{objref}) \)
3. If \( \text{Type}(x) \notin \{\text{XML, XMLList}\} \), throw a TypeError exception
4. Let \( l = \text{ToXMLList}(x) \)
5. Let \( r \) be a new XMLList with \( r.[[\text{TargetObject}]] = \text{null} \)
6. For \( i = 0 \) to \( l.[[\text{Length}]]-1 \)
   a. Add \( x[i] \) to the front of the scope chain
   b. Let \( \text{ref} \) be the result of evaluating Expression using the augmented scope chain of step 6a
   c. Let \( \text{match} = \text{ToBoolean}(\text{GetValue}(\text{ref})) \)
   d. Remove \( x[i] \) from the front of the scope chain
   e. If \( \text{match} == \text{true} \), call the \([\text{Append}]\) method of \( r \) with argument \( x[i] \)
7. Return \( r \)

The production \( \text{CallExpression} : \text{CallExpression} . ( \text{Expression} ) \) is evaluated in exactly the same manner, except that the contained \( \text{CallExpression} \) is evaluated in step 1.

10.3 Unary Operators

10.3.1 The delete Operator

This section is provided to describe the effects of the XML \([\text{Delete}]\) operators on the delete operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript delete operator beyond those specified by the XML and XMLList \([\text{Delete}]\) operators.

Syntax

E4X reuses the ECMAScript delete operator for deleting XML properties and XML attributes from XML objects and XMLLists. The syntax of the delete operator is described by the following production:

\[
\text{UnaryExpression} : \\
\text{delete} \ \text{UnaryExpression}
\]

Overview

When \( \text{UnaryExpression} \) evaluates to a Reference \( r \) with a base object of type XML, the delete operator removes the XML attributes or properties specified by the property name of \( r \) from the base object. When \( \text{UnaryExpression} \) evaluates to a Reference \( r \) with a base object of type XMLList, the delete operator removes the XML objects specified by the property name of \( r \) from the base object and the associated XML object. For example,

```javascript
delete order.customer.address; // delete the customer address
def order.customer.@id; // delete the customer ID
def order.item.price[0]; // delete the first item price
def order.item; // delete all the items
```

Semantics

E4X extends the semantics of the delete operator by providing more elaborate \([\text{Delete}]\) methods used when \( \text{UnaryExpression} \) evaluates to a value of type XML or XMLList (see sections 8.1.1.3 and 8.2.1.3 respectively).

10.3.2 The typeof Operator

Syntax

E4X reuses the syntax of the ECMAScript’s typeof operator for determining the types of XML and XMLList objects. The ECMAScript syntax for the typeof operator is described by the following production:

\[
\text{UnaryExpression} :
\]
E4X extends the semantics of the ECMAScript typeof operator for determining the types of XML and XMLList objects. When `UnaryExpression` evaluates to a value of type XML, the typeof operator returns the string "xml". When `UnaryExpression` evaluates to a value of type XMLList, the typeof operator returns the string "xmlList".

### Semantics

The production `UnaryExpression : typeof UnaryExpression` is evaluated as follows:

1. Let \( u \) be the result of evaluating `UnaryExpression`
2. If `Type(u)` is `Reference` and `GetBase(u)` is `null`, return "undefined"
3. Return a string determined by `Type(GetValue(u))` according to the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>&quot;boolean&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>&quot;number&quot;</td>
</tr>
<tr>
<td>String</td>
<td>&quot;string&quot;</td>
</tr>
<tr>
<td>XML</td>
<td>&quot;xml&quot;</td>
</tr>
<tr>
<td>XMLList</td>
<td>&quot;xml&quot;</td>
</tr>
<tr>
<td>Object (native and doesn't implement [[Call]])</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (native and implements [[Call]])</td>
<td>&quot;function&quot;</td>
</tr>
</tbody>
</table>

### 10.4 Additive Operators

#### Syntax

E4X reuses the syntax of the ECMAScript addition operator for concatenating two values of type XML or XMLList. The ECMAScript syntax for the addition operator is described by the following production:

\[
\text{AdditiveExpression} : \text{AdditiveExpression} + \text{MultiplicativeExpression}
\]

#### 10.4.1 The Addition Operator ( + )

##### Overview

E4X extends the semantics of the ECMAScript addition operator to perform either string concatenation, XML and XMLList concatenation or numeric addition depending on its arguments.

When both `AdditiveExpression` and `MultiplicativeExpression` evaluate to either an XML object or an XMLList, the addition operator starts by creating a new, empty XMLList as the return value. If the left operand evaluates to an XML object, it is added to the return value. If the left operand evaluates to an XMLList, each XML property of the XMLList is added to the return value in order. Likewise, if the right operand evaluates to an XML object, it is added to the return value. Otherwise, if it is an XMLList each XML property of the XMLList is added to the return value in order.

For example,
// create an XMLList containing the elements <name>, <age> and <hobby>
var employeedata = <name>Fred</name> + <age>28</age> + <hobby>skiing</hobby>;

// create an XMLList containing three item elements extracted from the order element
var myitems = order.item[0] + order.item[2] + order.item[3];

// create a new XMLList containing all the items in the order plus one new one
var newitems = order.item + <item><description>new item</description></item>

Note: Using the addition operator with operands of type XML and XMLList always results in an XMLList. When numeric addition of XML objects is desired, the operands must be explicitly coerced to Numbers. This may be accomplished by using the unary “+” operator or the Number conversion function. For example,

// add the prices of the first and third items in the order (coercion with unary +)
var totalPrice = +order.item[0].price + +order.item[2].price

// add the prices of the second and fourth items in the order (coercion using Number conversion function)
var totalPrice = Number(order.item[1].price) + Number(order.item[3].price)

Likewise, when string concatenation of XML objects is desired, at least one of the operands must be explicitly coerced to a String. This may be accomplished by concatenating them to the empty string ("") or using the String conversion function. For example,

// concatenate the street and the city of the customer's address (coercion with the empty string)
var streetcity = "" + order.customer.address.street + order.customer.address.city;

// concatenate the state and the zip of the customer's address (coercion using String conversion function)
var statezip = String(order.customer.address.state) + order.customer.address.zip;

Semantics
The production AdditiveExpression : AdditiveExpression + MultiplicativeExpression is evaluated as follows:

1. Let a be the result of evaluating AdditiveExpression
2. Let left = GetValue(a)
3. Let m be the result of evaluating MultiplicativeExpression
4. Let right = GetValue(m)
5. If (Type(left) ∈ {XML, XMLList}) and (Type(right) ∈ {XML, XMLList})
   a. Let l be a new XMLList
   b. Call the [[Append]] method of l with argument x
   c. Call the [[Append]] method of l with argument y
   d. Return l
6. Let pLeft = ToPrimitive(left)
7. Let pRight = ToPrimitive(right)
8. If Type(pLeft) is String or Type(pRight) is String
   a. Return the result of concatenating ToString(pLeft) and ToString(pRight)
9. Else
   a. Apply the addition operation to ToNumber(pLeft) and ToNumber(pRight) and return the result. See ECMAScript Edition 3, section 11.6.3 for details.

10.5 Equality Operators

10.5.1 The Abstract Equality Comparison Algorithm

Overview
E4X extends the abstract equality comparison algorithm defined by ECMAScript to enable equality comparisons involving QName and Namespace objects and the types XML and XMLList.

**Semantics**

The comparison \( x = y \), where \( x \) and \( y \) are values, produces **true** or **false**. This comparison is performed using the following steps:

1. If Type(\( x \)) is XMLList, call the [[Equals]] method of \( x \) with argument \( y \) and return the result
2. If Type(\( y \)) is XMLList, call the [[Equals]] method of \( y \) with argument \( x \) and return the result
3. If Type(\( x \)) is the same as Type(\( y \))
   a. If Type(\( x \)) is XML,
      i. If \((x.[[Class]]) \in \{"text", "attribute"\}\) and \((y.hasSimpleContent())\)
         or \((y.[[Class]]) \in \{"text", "attribute"\}\) and \((x.hasSimpleContent())\)
        l. Return the result of the comparison ToString(\( x \)) == ToString(\( y \))
   b. If Type(\( x \)) is Object and \( x.[[Class]] == "QName" \)
      i. If the result of the comparison \( x.uri == y.uri \) is **true** and the result of the comparison \( x.localName == y.localName \) is **true**, return **true**. Otherwise, return **false**
   c. If Type(\( x \)) is Object and \( x.[[Class]] == "Namespace" \), return the results of the comparison \( x.uri == y.uri \)
   d. If Type(\( x \)) is **undefined**, return **true**
   e. If Type(\( x \)) is **null**, return **true**
   f. If Type(\( x \)) is Number
      i. If \( x \) is NaN, return **false**
      ii. If \( y \) is NaN, return **false**
      iii. If \( x \) is the same number value as \( y \), return **true**
         iv. If \( x \) is +0 and \( y \) is -0, return **true**
         v. If \( x \) is -0 and \( y \) is +0, return **true**
         vi. Return **false**
   g. If Type(\( x \)) is String, then return **true** if \( x \) and \( y \) are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return **false**
   h. If Type(\( x \)) is boolean, return **true** if \( x \) and \( y \) are both **true** or both **false**. Otherwise, return **false**
      i. Return **true** if \( x \) and \( y \) refer to the same object or if they refer to objects joined to each other (ECMAScript Edition 3 Section 13.1.2). Otherwise, return **false**
4. If (Type(\( x \)) is XML) and \( x.hasSimpleContent() == true \) or (Type(\( y \)) is XML and \( y.hasSimpleContent() == true \))
   a. Return the result of the comparison ToString(\( x \)) == ToString(\( y \))
5. If \( x \) is **null** and \( y \) is **undefined**, return **true**
6. If \( x \) is **undefined** and \( y \) is **null**, return **true**
7. If Type(\( x \)) is Number and Type(\( y \)) is String, return the result of the comparison \( x == ToNumber(y) \)
8. If Type(\( x \)) is String and Type(\( y \)) is Number, return the result of the comparison ToNumber(\( x \)) == \( y \)
9. If Type(\( x \)) is Boolean, return the result of the comparison ToNumber(\( x \)) == \( y \)
10. If Type(\( y \)) is Boolean, return the result of the comparison \( x == ToNumber(y) \)
11. If Type(\( x \)) is either String or Number and Type(\( y \)) is Object, return the result of the comparison \( x == ToPrimitive(y) \)
12. If Type(\( x \)) is Object and Type(\( y \)) is either String or Number, return the result of the comparison ToPrimitive(\( x \)) == \( y \)
13. Return **false**.

**10.6 Assignment Operators**

**10.6.1 XML Assignment Operator**

This section is provided to describe the effects of the XML [[Put]] operator on the assignment operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript assignment operator beyond those specified by the XML and XMLList [[Put]] operators.

**Syntax**
E4X reuses the ECMAScript assignment operator to modify, replace and insert properties and XML attributes in an XML object. The ECMAScript syntax for the assignment operator is described by the following production:

\[
AssignmentExpression : \\
LeftHandSideExpression = AssignmentExpression
\]

**Overview**

The assignment operator begins by evaluating the `LeftHandSideExpression`, which resolves to a reference `r` consisting of a base object `parent` and a `property name`. If `parent` is an XML object, the assignment operator performs the steps described in section (see section 10.6.2 for the steps performed if `parent` is an XMLList).

If the `property name` begins with the character “@”, the XML assignment operator creates or modifies an XML attribute in the `parent`. If the named XML attribute already exists, the assignment operator modifies its value, otherwise it creates a new XML attribute with the given name and value. If `AssignmentExpression` evaluates to an XMLList, the value of the named attribute will be a space separated list of values (i.e., an XML attribute list) constructed by converting each value in the XMLList to a string and concatenating the results separated by spaces. If the `AssignmentExpression` does not evaluate to an XMLList, the value of the named attribute will be derived by evaluating the `AssignmentExpression` and calling `ToString` on the result. For example,

```javascript
order.item[1].@id = 123; // change the value of the id attribute on the second item
order.item[1].@newattr = "new value"; // add a new attribute to the second item
order.@allids = order.item.@id; // construct an attribute list containing all the ids in this order
```

If the `property name` is an array index, the XML assignment operator throws a `TypeError` exception. This operation is reserved for future versions of E4X.

If the `property name` does not begin with “@” and is not an array index, the XML assignment operator replaces, modifies or appends one or more XML objects in the `parent` by XML name. If only one XML object exists with the given name and the `AssignmentExpression` evaluates to an XML object or XMLList, the assignment operator replaces the identified XML object with the given value. If there are no XML properties with the given name, a new XML property with the given name and value is appended to the end of the `parent`. If more than one XML object exists with the given name and the `AssignmentExpression` evaluates to an XML object or XMLList, the assignment operator replaces the first XML property with a matching name with the given value and deletes the remaining XML properties with the given name, essentially replacing all the XML object with the given name with the given value. If the `AssignmentExpression` does not evaluate to a XML object or XMLList, the assignment operator calls `ToString` on the given value and replaces the properties (i.e., the content) of the appropriate XML object (as opposed to replacing the XML object itself). This provides a simple, intuitive syntax for setting the value of a named XML property to a primitive value. For example,

```javascript
item.price = 99.95; // change the price of the item
item.description = "Mobile Phone"; // change the description of the item
```

**Semantics**

E4X extends the semantics of the assignment operator by providing more elaborate `[[Put]]` methods used when `MemberExpression` evaluates to a value of type XML or XMLList (see sections 8.1.1.2 and 8.2.1.2 respectively).

### 10.6.2 XMLList Assignment Operator

This section is provided to describe the effects of the XMLList `[[Put]]` operator on the assignment operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript assignment operator beyond those provided by the XML and XMLList `[[Put]]` operators.

**Syntax**

E4X reuses the ECMAScript assignment operator to replace or append values to XMLLists and their associated XML objects. The ECMAScript syntax for the assignment operator is described by the following production:
AssignmentExpression:
LeftHandSideExpression = AssignmentExpression

Overview

The assignment operator begins by evaluating the LeftHandSideExpression, which resolves to a reference \( r \) consisting of a base object \( \text{parent} \) and a \text{property name}. If \( \text{parent} \) is an XMLList, the assignment operator performs the steps described in this section (see section 10.6.1 for the steps performed when \( \text{parent} \) is an XML object).

If the \text{property name} is not an array index, the XMLList assignment operator checks to see if this XMLList object contains only 1 item and that item is of type XML. If so, the XMLList assignment operator delegates its behaviour to the [[Put]] method of the XML object it contains (see section 8.1.1.2). This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one XML object. For example,

```xml
// set the name of the only customer in the order to Fred Jones
order.customer.name = "Fred Jones";

// replace all the hobbies for the only customer in the order
order.customer.hobby = "shopping";

// attempt to set the sale date of a single item.
order.item.saledate = "05-07-2002";

// replace all the employee's hobbies with their new favorite pastime
emps.employee.(@id == 3).hobby = "working";
```

In the first statement above, the expression “order.customer” returns an XMLList containing only one XML item. The expression “order.customer.name” implicitly converts this XMLList to an XML value and assigns the value “Fred Jones” to that value.

If the \text{property name} is an array index, the assignment operator replaces the property identified by \text{property name} in the XMLList or appends a new property if none exists with that \text{property name}. In addition, if the property identified is an XML value with a non-null \text{parent}, the XML value is also replaced in the context of its \text{parent}. If the AssignmentExpression evaluates to an XML value, the assignment operator replaces the value of the property identified by \text{property name} with a deep copy of the given XML object. If the AssignmentExpression evaluates to an XMLList, the assignment operator replaces the value of the property identified by \text{property name} with a deep copy of each item in the XMLList in order, effectively deleting the original property and inserting the contents of the XMLList in its place. If the AssignmentExpression does not evaluate to a value of type XML or XMLList, the assignment operator calls ToString on the given value and replaces the property at the given position with the result. Here are some examples,

```xml
// replace the first employee with George
e.employee[0] = <employee><name>George</name><age>27</age></employee>;

// add a new employee to the end of the employee list
e.employee[e.employee.length] = <employee><name>Frank</name></employee>;
```

Semantics

E4X extends the semantics of the assignment operator by providing more elaborate [[Put]] methods used when MemberExpression evaluates to a value of type XML or XMLList (see sections 8.1.1.2 and 8.2.1.2 respectively).
10.6.3 Compound Assignment (op=)
This section is provided to describe the effects of the XML and XMLList [[Get]] and [[Put]] operators on the compound assignment operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript compound assignment operator beyond those provided by the XML and XMLList [[Get]] and [[Put]] operators.

Syntax
E4X benefits from the compound assignment operator “+=” without requiring additional ECMAScript extensions. The syntax of the compound assignment “+=” is described by the following production:

\[
\text{AssignmentExpression} : \ \text{LeftHandSideExpression } += \text{ AssignmentExpression}
\]

Overview
When the left operand is an XML object, the “+=” operator has the effect of inserting one or more XML elements specified by the right operand just after the ordinal position of the left operand within its parent. For example,

```javascript
var e = <employees>
    <employee id="1"><name>Joe</name><age>20</age></employee>
    <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;
// insert employee 3 and 4 after the first employee
e.employee[0] += <employee id="3"><name>Fred</name></employee> +
    <employee id="4"><name>Carol</name></employee>;
```

Following the expressions above, the variable “e” would contain the XML value:

\[
\text{<employees>}
\text{<employee id="1"><name>Joe</name><age>20</age></employee>}
\text{<employee id="2"><name>Sue</name><age>30</age></employee>}
\text{<employee id="3"><name>Fred</name></employee> +}
\text{<employee id="4"><name>Carol</name></employee>}
\text{</employees>};
\]

When the left operand is an XMLList, the “+=” operator has the effect of appending one or more values specified by the right operand to the XMLList. If the last item in the XMLList is an XML object with a non-null parent, the “+=” operator also appends the items to the XML object referred to by parent just after the position of the last item in the XMLList. For example,

```javascript
var e = <employees>
    <employee id="1"><name>Joe</name><age>20</age></employee>
    <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;
// append employees 3 and 4 to the end of the employee list
e.employee += <employee id="3"><name>Fred</name></employee> +
    <employee id="4"><name>Carol</name></employee>;
```

Following the expressions above, the variable “e” would contain the XML value:
<employees>
    <employee id="1"><name>Joe</name><age>20</age></employee>
    <employee id="2"><name>Sue</name><age>30</age></employee>
    <employee id="3"><name>Fred</name></employee>
    <employee id="4"><name>Carol</name></employee>
</employees>

**Semantics**

E4X extends the semantics of the compound assignment operator by providing more elaborate [[Get]] and [[Put]] methods used when `MemberExpression` evaluates to a value of type XML or XMLList (see sections 8.1.1.1, 8.1.1.2, 8.2.1.1 and 8.2.1.2 respectively).
11 Statements
E4X extends the statements provided in ECMAScript with the following production:

Statement :
  DefaultXMLNamespaceStatement

11.1 The default xml namespace Statement

Syntax
E4X extends ECMAScript by adding a default xml namespace statement. The following production describes the syntax of the default xml namespace statement:

DefaultXMLNamespaceStatement :
  default xml namespace = Expression

Overview
The default xml namespace statement sets the value of the internal property [[DefaultNamespace]] of the variable object associated with the current execution context (see section 10 of ECMAScript Edition 3). If the variable object associated with the current execution context does not have an internal property [[DefaultNamespace]], the default xml namespace statement creates one. If the default xml namespace statement occurs inside a FunctionDeclaration, the default xml namespace is defined with function-local scope in that function. Otherwise, the default xml namespace is defined with global scope. Initially, the default xml namespace of the global scope is set to no namespace (section 12.1.1.1).

When the default xml namespace statement is executed, it evaluates the Expression, converts the result to a String s, creates a new Namespace object n as if by calling the constructor n = new Namespace("", s), and sets the default XML namespace associated with the current execution context to n. Unqualified XML element names following the default xml namespace declaration in the current scope will be associated with the default xml namespace specified by Expression. For example,
// declare some namespaces and a default namespace for the current scope
var soap = new Namespace("http://schemas.xmlsoap.org/soap/envelope/");
var stock = new Namespace("http://mycompany.com/stocks");
default xml namespace = soap;  // alternately, may specify full URI

// Create an XML initializer in the default (i.e., soap) namespace
var message = <Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <Body>
    <m:GetLastTradePrice xmlns:m="http://mycompany.com/stocks">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </Body>
</Envelope>

// extract the soap encoding style using a QualifiedIdentifier (unqualified attributes are in no namespace)
var encodingStyle = message.@soap::encodingStyle;

// extract the body from the soap message using the default namespace
var body = message.Body;

// change the stock symbol using the default namespace and qualified names
message.Body.stock::GetLastTradePrice.stock::symbol = "MYCO";

Semantics

The production DefaultXMLNamespaceStatement : default xml namespace = Expression is evaluated as follows:

1. Let uriRef be the result of evaluating Expression
2. Let uri = GetValue(uriRef)
3. Let namespace be a new Namespace Object, created as if by calling the constructor new Namespace("", uri)
4. Let varObj be the variable object associated with the current execution context (see section 10.1.3 of ECMAScript Edition 3)
5. Let varObj.[[DefaultNamespace]] = namespace

11.1.1 GetDefaultNamespace ( )

Overview

The GetDefaultNamespace abstract operator is used in this specification for obtaining the value of the default xml namespace. This operator is not part of the language. It is defined here to aid the specification of the language.

Semantics

When the internal GetDefaultNamespace method is called, the following steps are taken:

1. While (there are more objects on the scope chain)
   a. Let o be the next object on the scope chain
      Note: on the first iteration, o will be the first object on the scope chain
   b. If o has the internal property [[DefaultNamespace]], return o.[[DefaultNamespace]]
2. Throw a TypeError exception
   Note: this step should not occur because the global object has a [[DefaultNamespace]] property
11.2 The for-in Statement

Syntax

E4X extends the semantics of the ECMAScript for-in statement used for iterating over the properties of an object. The syntax of the for-in statement is specified by the following productions:

\[
\text{IterationStatement} : \\
\quad \text{for ( LeftHandSideExpression in Expression ) Statement} \\
\quad \text{for ( var VariableDeclarationNoLn in Expression ) Statement}
\]

Overview

The for-in statement evaluates the Expression and iterates through each property of the resulting object. For each property, the for-in statement assigns the name of the property to the variable identified by LeftHandSideExpression or VariableDeclarationNoLn and evaluates the Statement. For example:

```javascript
// print all the employee names
for (var n in e_.name) {
    print("Employee name: " + e_.name[n]);
}

// print each child of the first item
for (var child in order.item[0].*) {
    print("item child: " + order.item.[0].*[child]);
}
```

When the Expression evaluates to an XML object, the for-in statement converts the XML object to an XMLList and iterates over the resulting XMLList. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value.

Semantics

The production \text{IterationStatement} : for ( LeftHandSideExpression in Expression ) Statement is evaluated as follows:

1. Let \text{ref} be the result of evaluating \text{Expression}
2. Let \text{e} = \text{GetValue(}\text{ref})
3. if Type(\text{e}) is XML
   a. Let \text{l} = \text{ToXMLList(}\text{e})
4. else if Type(\text{e}) is not XMLList
   a. Let \text{l} = \text{ToObject(}\text{e})
5. Let \text{V} = \text{empty}
6. While (\text{l} has more properties)
   a. Let \text{p} be the next property of \text{l} (see notes below)
   b. If \text{p} does not have the DontEnum attribute
      i. Let \text{i} be the result of evaluating \text{LeftHandSideExpression}
      ii. Let \text{name} be the name of property \text{p}
      iii. PutValue(\text{i}, \text{name})
      iv. Let \text{s} be the result of evaluating \text{Statement}
      v. If \text{s}.value is not empty, let \text{V} = \text{s}.value
     vi. If \text{s}.type is break and \text{s}.target is in the current label set, return (normal, \text{V}, \text{empty})
     vii. If \text{s}.type is not continue or \text{s}.target is not in the current label set
        1. If \text{s} is an abrupt completion, return \text{s}
7. Return (normal, \text{V}, \text{empty})

The production \text{IterationStatement} : for ( var VariableDeclarationNoLn in Expression ) Statement is evaluated as follows:
1. Let varRef be the result of evaluating VariableDeclarationNoLn
2. Let ref be the result of evaluating Expression
3. Let $e = \text{GetValue(ref)}$
4. If Type($e$) is XML
   a. Let $l = \text{ToXMLList}(e)$
5. else if Type($e$) is not XMLList
   a. Let $l = \text{ToObject}(e)$
6. Let $V = \text{empty}$
7. While ($l$ has more properties)
   a. Let $p$ be the next property of $l$ (see notes below)
   b. If $p$ does not have the DontEnum attribute
      i. Let $i$ be the result of evaluating varRef as if it were an identifier (See section 11.1.2 of ECMAScript Edition 3)
      ii. Let name be the name of property $p$
      iii. PutValue($i$, name)
      iv. Let $s$ be the result of evaluating Statement
      v. If $s$.value is not empty, let $V = s$.value
      vi. If $s$.type is break and $s$.target is in the current label set, return (normal, $V$, empty)
      vii. If ($s$.type is not continue) or ($s$.target is not in the current label set)
         1. If $s$ is an abrupt completion, return $s$
8. Return (normal, $V$, empty)

The order of enumeration is defined by the object (steps 6 and 6a in the first algorithm and steps 7 and 7a in the second algorithm). When $e$ evaluates to a value of type XML or XMLList, properties are enumerated in ascending sequential order according to their numeric property names (i.e., document order for XML objects).

The mechanics of enumerating the properties (steps 7 and 7a in the first algorithm, steps 8 and 8a in the second) is implementation dependent. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. Enumerating the properties of an object includes enumerating properties of its prototype and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is "shadowed" because some previous object in the prototype chain has a property with the same name.

11.3 The for-each-in Statement

Syntax

E4X extends ECMAScript by adding a for-each-in statement for iterating over the property values of an object. The syntax of the for-each-in statement is specified by the following productions:

\[
\text{IterationStatement} : \\
\quad \text{for each (} \text{LeftHandSideExpression in Expression} \) \text{ Statement} \\
\quad \text{for each (} \text{var VariableDeclarationNoLn in Expression} \) \text{ Statement}
\]

Overview

The for-each-in statement evaluates the Expression and iterates through each property of the resulting object. For each property, the for-each-in statement assigns the value of the property to the variable identified by LeftHandSideExpression or VariableDeclarationNoLn and evaluates the Statement. For example:
// print all the employee names
for each (var n in e..name) {
    print ("Employee name: " + n);
}

// print each child of the first item
for each (var child in order.item[0].*) {
    print("item child: " + child);
}

In the first for-each-in statement above, the expression “e..name” returns an XMLList containing all of the descendant XML properties of the XML object “e” with the name “name”. The for-each-in statement iterates through each property of the resulting XMLList object in order. For each XML property in the list, it assigns the value of the XML property to the variable “n” and executes the code nested in curly braces. Similarly, in the second for-each-in statement above, the expression “order.item[0].*” returns an XMLList containing all of the children of the first XML object named “item” from the XML object named “order”. The for-each-in statement iterates through each property of the XMLList object in order assigning the value of the XML property to the variable “child” and executing the code nested in curly braces.

When the Expression evaluates to an XML object, the for-each-in statement converts the XML object to an XMLList and iterates over the resulting XMLList. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value.

Note: The for-each-in statement behaves differently than the for-in statement. In particular, it assigns the loop variable over the range of the object rather than the domain of the object. I.e., the for-each-in statement binds the loop variable to the property values of the given object rather than the property names.

**Semantics**

The production **IterationStatement** : for each ( **LeftHandSideExpression** in **Expression** ) **Statement** is evaluated as follows:

1. Let ref be the result of evaluating Expression
2. Let e = GetValue(ref)
3. if Type(e) is XML
   a. Let l = ToXMLList(e)
4. else if Type(e) is not XMLList
   a. Let l = ToObject(e)
5. Let V = empty
6. While (l has more properties)
   a. Let p be the next property of l (see notes below)
   b. If p does not have the DontEnum attribute
      i. Let i be the result of evaluating LeftHandSideExpression
      ii. Let value be the value of property p
      iii. PutValue(i, value)
   iv. Let s be the result of evaluating Statement
   v. If s.value is not empty, let V = s.value
   vi. If s.type is break and s.target is in the current label set, return (normal, V, empty)
   vii. If (s.type is not continue) or (s.target is not in the current label set)
      1. If s is an abrupt completion, return s
7. Return (normal, V, empty)

The production **IterationStatement** : for each ( var **VariableDeclarationNoLn** in **Expression** ) **Statement** is evaluated as follows:

1. Let varRef be the result of evaluating VariableDeclarationNoLn
2. Let ref be the result of evaluating Expression
3. Let e = GetValue(ref)
4. if Type(e) is XML
   a. Let l = ToXMLList(e)
5. else if Type(e) is not XMLList
   a. Let l = ToObject(e)
6. Let V = empty
7. While (l has more properties)
   a. Let p be the next property of l (see notes below)
   b. If p does not have the DontEnum attribute
      i. Let i be the result of evaluating varRef as if it were an identifier (See section 11.1.2 of ECMAScript Edition 3)
      ii. Let value be the value of property p
      iii. PutValue(i, value)
      iv. Let s be the result of evaluating Statement
      v. If s.value is not empty, let V = s.value
      vi. If (s.type is break and s.target is in the current label set), return (normal, V, empty)
      vii. If (s.type is not continue) or (s.target is not in the current label set)
          1. If s is an abrupt completion, return s
8. Return (normal, V, empty)

The order of enumeration is defined by the object (steps 6 and 6a in the first algorithm and steps 7 and 7a in the second algorithm). When e evaluates to a value of type XML or XMLList, properties are enumerated in ascending sequential order according to their numeric property names (i.e., document order for XML objects).

The mechanics of enumerating the properties (steps 7 and 7a in the first algorithm, steps 8 and 8a in the second) is implementation dependent. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it may not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. Enumerating the properties of an object includes enumerating properties of its prototype and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is "shadowed" because some previous object in the prototype chain has a property with the same name.
12 Native E4X Objects

E4X adds two native objects to ECMAScript, the native XML object and the native XMLList object. In addition, E4X adds new properties to the global object.

12.1 The Global Object

12.1.1 Internal Properties of the Global Object

E4X extends ECMAScript by adding the following internal properties to the global object.

12.1.1.1 [[DefaultNamespace]]

Overview

Initially, the global object has an internal property `[[DefaultNamespace]]` with its value set to a Namespace object representing no namespace, created as if by calling the Namespace constructor with no arguments. Consequently, unless otherwise specified using the default xml namespace statement (see section 11.1), unqualified names used to specify properties of XML objects will match XML properties in no namespace.

12.1.2 Function Properties of the Global Object

E4X extends ECMAScript by adding the following function properties to the global object.

12.1.2.1 isXMLName (value)

Overview

The `isXMLName` function examines the given `value` and determines whether it is a valid XML name that can be used as an XML element or attribute name. If so, it returns `true`, otherwise it returns `false`.

Semantics

When the `isXMLName` function is called with one parameter `value`, the following steps are taken:

1. Let `q` be a new QName created as if by calling the constructor `new QName(value)` and if a `TypeError` exception is thrown, return `false`
2. If `q.localName` does not match the production `NCName`, return `false`
3. Return `true`

Where the production `NCName` is defined in Section 2 of the Namespaces in XML specification.

12.1.3 Constructor Properties of the Global Object

E4X extends ECMAScript by adding the following constructor properties to ECMAScript.

12.1.3.1 Namespace ( . . . )

See section 12.2.1 and 12.2.

12.1.3.2 QName ( . . . )

See section 12.3.1 and 12.3.2.

12.1.3.3 XML ( . . . )

See sections 12.4.1 and 12.4.2.

12.1.3.4 XMLList ( . . . )

See section 12.5.1 and 1.
12.2 Namespace Objects

Namespace objects represent XML namespaces and provide an association between a namespace prefix and a Unique Resource Identifier (URI). The prefix is either the undefined value or a string value that may be used to reference the namespace within the lexical representation of an XML value. When an XML object containing a namespace with an undefined prefix is converted to a string, the implementation will automatically generate a prefix. The URI is a string value used to uniquely identify the namespace.

12.2.1 The Namespace Constructor Called as a Function

Syntax

Namespace ( )
Namespace ( uriValue )
Namespace ( prefixValue , uriValue )

Overview

If the Namespace constructor is called as a function with exactly one argument that is a Namespace object, the argument is returned unchanged. Otherwise, a new Namespace object is created and returned as if the same arguments were passed to the object creation expression new Namespace ( ... ). See section 12.2.2.

Semantics

When Namespace is called as a function with a no arguments, one argument uriValue, or two arguments prefixValue and uriValue, the following steps are taken:

1. If (prefixValue is not specified and Type(uriValue) is Object and uriValue.[[Class]] === "Namespace")
   a. Return uriValue
2. Create and return a new Namespace object exactly as if the Namespace constructor had been called with the same arguments (section 12.2.2).

12.2.2 The Namespace Constructor

Syntax

new Namespace ( )
new Namespace ( uriValue )
new Namespace ( prefixValue , uriValue )

Overview

When Namespace is called as part of a new expression, it is a constructor and creates a new Namespace object.

The [[Prototype]] property of the newly constructed object is set to the original Namespace prototype object, the one that is the initial value of Namespace.prototype (section 12.2.3.1). The [[Class]] property of the newly constructed object is set to “Namespace”.

When no arguments are specified, the namespace uri and the prefix are set to the empty string. A namespace with uri set to the empty string represents no namespace. No namespace is used in XML objects to explicitly specify that a name is not inside a namespace and may never be associated with a prefix other than the empty string.

When only the uriValue argument is specified and uriValue is a Namespace object, a copy of the uriValue is returned. When only the uriValue is specified and it is the empty string, the prefix is set to the empty string. In all other cases where only the uriValue is specified, the namespace prefix is set to the undefined value.

When the prefixValue argument is specified and set to the empty string, the Namespace is called a default namespace. Default namespaces are used in XML objects to implicitly specify the namespace of qualified names that do not specify a qualifier.

Semantics
When the Namespace constructor is called with a no arguments, one argument `uriValue` or two arguments `prefixValue` and `uriValue`, the following steps are taken:

1. Create a new Namespace object `n`
2. If `prefixValue` is not specified and `uriValue` is not specified
   a. Let `n.prefix` be the empty string
   b. Let `n.uri` be the empty string
3. Else if `prefixValue` is not specified
   a. If Type(`uriValue`) is Object and `uriValue.[[Class]]` == "Namespace"
      i. Let `n.prefix` = `uriValue.prefix`
      ii. Let `n.uri` = `uriValue.uri`
   b. Else if Type(`uriValue`) is Object and `uriValue.[[Class]]` == "QName" and `uriValue.uri` is not null
      i. Let `n.uri` = `uriValue.uri`
      Note: implementations that preserve prefixes in qualified names may also set `n.prefix` = `uriValue.[[Prefix]]`
   c. Else
      i. Let `n.uri` = ToString(`uriValue`)
      ii. If (`n.uri` is the empty string), let `n.prefix` be the empty string
      iii. Else `n.prefix` = `undefined`
4. Else
   a. if Type(`uriValue`) is Object and `uriValue.[[Class]]` == "QName" and `uriValue.uri` is not null
      i. Let `n.uri` = `uriValue.uri`
   b. Else
      i. Let `n.uri` = ToString(`uriValue`)
   c. If `n.uri` is the empty string
      i. If `prefixValue` is `undefined` or ToString(`prefixValue`) is the empty string
         i. Let `n.prefix` be the empty string
      ii. Else throw a `TypeError` exception
   d. Else if `prefixValue` is `undefined`, let `n.prefix` = `undefined`
   e. Else if isXMLName(`prefixValue`) == `false`
      i. Let `n.prefix` = `undefined`
   f. Else let `n.prefix` = ToString(`prefixValue`)
5. Return `n`

12.2.3 Properties of the Namespace Constructor
The value of the internal `[[Prototype]]` property of the Namespace constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 2), the Namespace constructor has the following properties.

12.2.3.1 Namespace.prototype
The initial value of the Namespace.prototype property is the Namespace prototype object (section 12.2.4).

This property has the attributes `{ DontEnum, DontDelete, ReadOnly }`.

12.2.4 Properties of the Namespace Prototype Object (Built-in Methods)
The Namespace prototype object is itself a Namespace object (its `[[Class]]` is “Namespace”) with its `uri` and `prefix` properties set to the empty string.

The value of the internal `[[Prototype]]` property of the Namespace prototype object is the Object prototype object (section 15.2.3.1 of ECMAScript Edition 3).

12.2.4.1 Namespace.prototype.constructor
The initial value of the Namespace.prototype.constructor is the built-in Namespace constructor.
12.2.4.2 Namespace.prototype.toString()

Overview

The toString() method returns a string representation of this Namespace object.

The toString function is not generic. It throws a TypeError exception if its this value is not a Namespace object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

Semantics

When the toString method of a Namespace object n is called with no arguments, the following step is taken:

1. if Type(n) is not Object or n.[[Class]] is not equal to "Namespace", throw a TypeError exception
2. Return n.uri

12.2.5 Properties of Namespace Instances

Namespace instances inherit properties from the Namespace prototype object and also have a prefix property and a uri property.

12.2.5.1 prefix

The value of the prefix property is either the undefined value or a string value. When the value of the prefix property is the empty string, the Namespace is called a default namespace. Default namespaces are used in XML objects to determine the namespace of names that do not specify a qualifier.

This property has the attributes { DontDelete, ReadOnly }

12.2.5.2 uri

The value of the uri property is a string value. When the value of the uri property is the empty string, the Namespace represents the unnamed namespace. The unnamed namespace is used in XML objects to explicitly specify that a name is not inside a namespace.

This property has the attributes { DontDelete, ReadOnly }

12.3 QName Objects

QName objects are used to represent qualified names of XML elements and attributes. Each QName object has a local name of type string and a namespace URI of type string or null. When the namespace URI is null, this qualified name matches any namespace.

Implementations may include an internal [[Prefix]] property that is not directly visible to E4X users. When a QName object is created using a Namespace object argument, the internal [[Prefix]] property may be used to preserve the prefix of the Namespace object. If no namespace prefix was specified for the associated Namespace, the [[Prefix]] property may be undefined.

A value of type QName may be specified using a QualifiedIdentifier. If the QName of an XML element is specified without identifying a namespace (i.e., as an unqualified identifier), the uri property of the associated QName will be set to the in-scope default namespace (section 11.1). If the QName of an XML attribute is specified without identifying a namespace, the uri property of the associated QName will be the empty string representing no namespace.

12.3.1 The QName Constructor Called as a Function

Syntax

QName ( Name )
QName ( Namespace , Name )

Overview
If the QName constructor is called as a function with exactly one argument that is a QName object, the argument is returned unchanged. Otherwise, a new QName object is created and returned as if the same arguments were passed to the object relation expression `new QName ( ... )`. See section 12.3.2.

Semantics

When the QName function is called the following step is taken.

1. If `Namespace` is not specified and `Type(Name)` is Object and `Name.[[Class]] == “QName”`
   a. Return `Name`
2. Create and return a new QName object exactly as if the QName constructor had been called with the same arguments (section 12.3.2).

12.3.2 The QName Constructor

Syntax

```
new QName ( Name )
new QName ( Namespace , Name )
```

Overview

When QName is called as part of a new expression, it is a constructor and creates a new QName object.

The `[[Prototype]]` property of the newly constructed object is set to the original QName prototype object, the one that is the initial value of QName.prototype (section 12.3.3.1). The `[[Class]]` property of the newly constructed object is set to “QName”.

If `Name` is a QName and `Namespace` is not specified, the QName constructor returns a copy of the given `Name`.

When both the `Namespace` and `Name` arguments are specified, the `localName` property of the newly created object is set according to the given `Name` and the `uri` property of the newly created object is set according to the `Namespace` argument. If `Name` is a QName object, the `localName` of the newly created QName will be equal to the `localName` of `Name`. If the `Namespace` argument is a Namespace object, the `uri` property of the newly created object is set to the `uri` property of the Namespace object. If the `Namespace` argument is `null`, the `uri` property of the newly created object will be `null`, meaning it will match names in any namespace.

Semantics

When the QName constructor is called with a one argument `Name` or two arguments `Namespace` and `Name` the following steps are taken:

1. If (Type(Name) is Object and Name.[[Class]] == "QName")
   a. If (Namespace is not specified), return a copy of Name
   b. Else let Name = Name.localName
2. Let Name = ToString(Name)
3. If (Namespace is `undefined` or not specified)
   a. If Name = "*"
      i. Let Namespace = `null`
   b. Else
      i. Let Namespace = GetDefaultNamespace()
4. Let q be a new QName with q.localName = Name
5. If Namespace == `null`
   a. Let q.uri = `null`
   Note: implementations that preserve prefixes in qualified names may also set q.[[Prefix]] to `undefined`
6. Else
   a. Let Namespace be a new Namespace created as if by calling the constructor new Namespace(Namespace)
Let $q.uri = \text{Namespace}\_\text{uri}$

Note: implementations that preserve prefixes in qualified names may also set $q.[[\text{Prefix}]]$ to $\text{Namespace}\_\text{prefix}$

Return $q$

12.3.3 Properties of the QName Constructor

The value of the internal $[[\text{Prototype}]]$ property of the QName constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 2), the QName constructor has the following properties.

12.3.3.1 QName.prototype

The initial value of the QName.prototype property is the QName prototype object (section 12.3.4).

This property has the attributes { DontEnum, DontDelete, ReadOnly }.

12.3.4 Properties of the QName Prototype Object

The QName prototype object is itself a QName object (its $[[\text{Class}]]$ is “QName”) with its $uri$ and $localName$ properties set to the empty string.

The value of the internal $[[\text{Prototype}]]$ property of the QName prototype object is the Object prototype object (section 15.2.3.1 of ECMAScript Edition 3).

12.3.4.1 QName.prototype.constructor

The initial value of the QName.prototype.constructor is the built-in QName constructor.

12.3.4.2 QName.prototype.toString()

Overview

The toString method returns a string representation of this QName object.

The toString function is not generic. It throws a TypeError exception if its this value is not a QName object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

Semantics

When the toString method of a QName object $n$ is called with no arguments, the following steps are taken:

1. If Type($n$) is not Object or $n.[[\text{Class}]]$ is not equal to "QName", throw a TypeError exception
2. Let $s$ be the empty string
3. If $n.uri$ is not the empty string
   a. If $n.uri$ === null, let $s$ be the string "*::"
   b. Else let $s$ be the result of concatenating $n.uri$ and the string "::"
4. Let $s$ be the result of concatenating $s$ and $n.localName$
5. Return $s$

12.3.5 Properties of QName Instances

QName instances inherit properties from the QName prototype object and also have a $uri$ property, a $localName$ property and an optional internal $[[\text{Prefix}]]$ property that may be used by implementations that preserve prefixes in qualified names.

12.3.5.1 localName

The value of the $localName$ property is a value of type string. When the value of the $localName$ property is "*" it represents a wildcard that matches any name.

This property shall have the attributes { DontDelete, ReadOnly }
12.3.5.2 uri
The value of the uri property is null or a value of type string identifying the namespace of this QName. When the value of the uri property is the empty string, this QName is said to be in no namespace. No namespace is used in XML objects to explicitly specify that a name is not inside a namespace. When the value of the uri property is null, this QName will match names in any namespace.

This property shall have the attributes { DontDelete, ReadOnly }

12.3.5.3 [[Prefix]]
The [[Prefix]] property is an optional internal property that is not directly visible to users. It may be used by implementations that preserve prefixes in qualified names. The value of the [[Prefix]] property is a value of type string or undefined. If the [[Prefix]] property is undefined, the prefix associated with this QName is unknown.

12.3.5.4 [[GetNamespace]] ( [ InScopeNamespaces ] )
Overview
The [[GetNamespace]] method is an internal method that returns a Namespace object with a URI matching the URI of this QName. InScopeNamespaces is an optional parameter. If InScopeNamespaces is unspecified, it is set to the empty set. If one or more Namespaces exists in InScopeNamespaces with a URI matching the URI of this QName, one of the matching Namespaces will be returned. If no such namespace exists in InScopeNamespaces, [[GetNamespace]] creates and returns a new Namespace with a URI matching that of this QName. For implementations that preserve prefixes in QNames, [[GetNamespace]] may return a Namespace that also has a matching prefix. The input argument InScopeNamespaces is a set of Namespace objects.

Semantics
When the [[GetNamespace]] method of a QName q is called with no arguments or one argument InScopeNamespaces, the following steps are taken:

1. if q.uri is null, throw a TypeError exception
   Note: the exception above should never occur due to the way [[GetNamespace]] is called in this specification
2. If InScopeNamespaces was not specified, let InScopeNamespaces = { }
3. Find a Namespace ns in InScopeNamespaces, such that ns.uri == q.uri. If more than one such Namespace ns exists, the implementation may choose one of the matching Namespaces arbitrarily.
   Note: implementations that preserve prefixes in qualified names may additionally constrain ns, such that ns.prefix == q.[[Prefix]]
4. If no such a namespace ns exists
   a. Let ns be a new namespace created as if by calling the constructor new Namespace(q.uri)
      Note: implementations that preserve prefixes and qualified names may create the new namespaces as if by calling the constructor Namespace(q.[[Prefix]], q.uri)
5. return ns

12.4 XML Objects

12.4.1 The XML Constructor Called as a Function
Syntax
XML ( [ value ] )
Overview
When XML is called as a function rather than as a constructor, it performs a type conversion. If no argument is provided, the XML function returns an XML object representing an empty text node.

Semantics
When the XML function is called with no arguments or with one argument \textit{value}, the following step is taken.

1. If \textit{value} is \texttt{null}, \texttt{undefined} or not supplied, let \textit{value} be the empty string
2. Return \texttt{ToXML(value)}

Note: The \texttt{ToXML} operator defines a mechanism for constructing an XML object from an implementation of the W3C information set (e.g., a W3C DOM node). E4X implementations may expose this functionality to users via the XML constructor; however, this is not required for conformance with E4X. See section 9.3.2 for additional information.

**12.4.2 The XML Constructor**

**Syntax**

\texttt{new XML ( [ value ] )}

**Overview**

When XML is called as part of a new expression, it is a constructor and may create a new XML object. When the XMLList constructor is called with no arguments, it returns an XML object representing an empty text node.

**Semantics**

When the XML constructor is called with no arguments or a single argument \textit{value}, the following steps are taken:

1. If \textit{value} is \texttt{null}, \texttt{undefined} or not supplied, let \textit{value} be the empty string
2. Let \textit{x} = \texttt{ToXML(value)}
3. If Type(\textit{value}) \in \{XML, XMLList, W3C XML Information Item\}
   a. Return the result of calling the \texttt{[[DeepCopy]]} method of \textit{x}
4. Return \textit{x}

**12.4.3 Properties of the XML Constructor**

The value of the internal \texttt{[[Prototype]]} property of the XML constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 1), the XML constructor has the following properties:

**12.4.3.1 XML.prototype**

The initial value of the XML.prototype property is the XML prototype object (section 12.4.3.7).

This property has the attributes \{ DontEnum, DontDelete, ReadOnly \}.

**12.4.3.2 XML.ignoreComments**

The initial value of the ignoreComments property is \texttt{true}. If ignoreComments is \texttt{true}, XML comments are ignored when constructing new XML objects. This property has the attributes \{ DontEnum, DontDelete \}.

**12.4.3.3 XML.ignoreProcessingInstructions**

The initial value of the ignoreProcessingInstructions property is \texttt{true}. If ignoreProcessingInstructions is \texttt{true}, XML processing instructions are ignored when constructing new XML objects. This property has the attributes \{ DontEnum, DontDelete \}.

**12.4.3.4 XML.ignoreWhitespace**

The initial value of the ignoreWhitespace property is \texttt{true}. If ignoreWhitespace is \texttt{true}, insignificant whitespace characters are ignored when processing constructing new XML objects. When elements tags and/or embedded expressions are separated only by whitespace characters, those whitespace characters are defined to be insignificant. Whitespace characters are defined
to be space (\u0020), carriage return (\u000D), line feed (\u000A) and tab (\u0009). This property has the attributes {DontEnum, DontDelete}.

12.4.3.5 XML.prettyPrinting

The initial value of the prettyPrinting property is true. If prettyPrinting is true, the ToString and ToXMLString operators will normalize whitespace characters between certain tags to achieve a uniform and aesthetic appearance. This property has the attributes {DontEnum, DontDelete}.

12.4.3.6 XML.prettyIndent

The initial value of the prettyIndent property is 2. If the prettyPrinting property of the XML constructor is true, the ToString and ToXMLString operators will normalize whitespace characters between certain tags to achieve a uniform and aesthetic appearance. Certain child nodes will be indented relative to their parent node by the number of spaces specified by prettyIndent. This property has the attributes {DontEnum, DontDelete}.

12.4.3.7 XML.settings( )

Overview

The settings method is a convenience method for managing the collection of global XML settings stored as properties of the XML constructor (sections 12.4.3.2 through 12.4.3.6). It returns an object containing the properties of the XML constructor used for storing XML settings. This object may later be passed as an argument to the setSettings method to restore the associated settings. For example,

```javascript
// Create a general purpose function that may need to save and restore XML settings
function getXMLCommentsFromString(xmlString) {
    // save previous XML settings and make sure comments are not ignored
    var settings = XML.settings();
    XML.ignoreComments = false;

    var comments = XML(xmlString).comment();

    // restore settings and return result
    XML.setSettings(settings);
    return comments;
}
```

Semantics

When the settings method of the XML constructor is called, the following steps are taken:

1. Let s be a new Object created as if by calling the constructor new Object()
2. Let s.ignoreComments = XML.ignoreComments
3. Let s.ignoreProcessingInstructions = XML.ignoreProcessingInstructions
4. Let s.ignoreWhitespace = XML.ignoreWhitespace
5. Let s.prettyPrinting = XML.prettyPrinting
6. Let s.prettyIndent = XML.prettyIndent
7. Return s

12.4.3.8 XML.setSettings([ Settings ])

The setSettings method is a convenience method for managing the collection of global XML settings stored as properties of the XML constructor (sections 12.4.3.2 through 12.4.3.6). It may be used to restore a collection of XML settings captured earlier using the associated settings method. When called with a single argument settings, the setSettings method copies the properties of the XML constructor used for storing XML settings from the settings object. When called with no arguments, the setSettings method restores the default XML settings.
Semantics

When the setSettings method of the XML constructor is called with no arguments or with a single argument settings, the following steps are taken:

1. If settings is null, undefined or not provided
   a. Let XML.ignoreComments = true
   b. Let XML.ignoreProcessingInstructions = true
   c. Let XML.ignoreWhitespace = true
   d. Let XML.prettyPrinting = true
   e. Let XML.prettyIndent = 2
2. else if Type(settings) is Object
   a. If Type(settings.ignoreComments) is Boolean, i. Let XML.ignoreComments = settings.ignoreComments
   b. If Type(settings.ignoreProcessingInstructions) is Boolean, i. Let XML.ignoreProcessingInstructions = settings.ignoreProcessingInstructions
   c. If Type(settings.ignoreWhitespace) is Boolean, i. Let XML.ignoreWhitespace = settings.ignoreWhitespace
   d. If Type(settings.prettyPrinting) is Boolean, i. Let XML.prettyPrinting = settings.prettyPrinting
   e. If Type(settings.prettyIndent) is Number, i. Let XML.prettyIndent = settings.prettyIndent
3. Return

12.4.3.9 XML.defaultSettings ( )

The defaultSettings method is a convenience method for managing the collection of global XML settings stored as properties of the XML constructor (sections 12.4.3.2 through 12.4.3.6). It may be used to obtain an object containing the default XML settings. This object may be inspected to determine the default settings or be passed as an argument to the setSettings method to restore the default XML settings.

Semantics

When the defaultSettings method of the XML constructor is called with no arguments, the following steps are taken:

1. Let s be a new Object created as if by calling the constructor new Object()
2. Let s.ignoreComments = true
3. Let s.ignoreProcessingInstructions = true
4. Let s.ignoreWhitespace = true
5. Let s.prettyPrinting = true
6. Let s.prettyIndent = 2
7. Return s

12.4.3.10 [[HasInstance]] ( V )

Overview

The XML constructor has a more elaborate [[HasInstance]] method than other function objects. The XML constructor [[HasInstance]] method is defined such that given an XML object or an XMLList object x, the expression x instanceof XML will return true. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value.

Semantics

When the [[HasInstance]] method of the XML constructor object F is called with value V, the following steps are taken:

1. If V is not an object, return false
2. Let xmlProto = XML.prototype
3. Let listProto = XMLList.prototype
4. if (xmlProto is not an object) or (listProto is not an object), throw a TypeError exception
5. Let objProto = V.[[Prototype]]
6. while (objProto is not null)
   a. if xmlProto and objProto refer to the same object or if they refer to objects joined to each other (section 13.1.2 of ECMAScript Edition 3), return true
   b. if listProto and objProto refer to the same object or if they refer to objects joined to each other (section 13.1.2 of ECMAScript Edition 3), return true
   c. Let objProto = objProto.[[Prototype]]
7. Return false

12.4.4 Properties of the XML Prototype Object (Built-in Methods)

Each value of type XML has a set of built-in methods available for performing common operations. These built-in methods are properties of the XML prototype object and are described in the following sections.

The XML prototype object is itself an XML object (its [[Class]] property is "text") whose value is the empty string.

The value of the internal [[Prototype]] property of the XML prototype object is the Object prototype object. (section 15.2.3.1 of ECMAScript Edition 3).

None of the built-in functions defined on XML.prototype are generic. They throw a TypeError exception if the this value is not an XML object. Therefore, they cannot be transferred to other kinds of objects for use as a method.

12.4.4.1 XML.prototype.constructor

The initial value of the XML.prototype.constructor is the built-in XML constructor.

12.4.4.2 XML.prototype.addNamespace (namespace)

Overview

The addNamespace method adds a namespace declaration to the in scope namespaces for this XML object and returns this XML object. If the in scope namespaces for the XML object already contains a namespace with a prefix matching that of the given parameter, the prefix of the existing namespace is set to undefined.

Semantics

When the addNamespace method of an XML object x is called with one parameter namespace, the following step is taken:

1. Let ns a Namespace constructed as if by calling the function Namespace(namespace)
2. Call the [[AddInScopeNamespace]] method of x with parameter ns
3. Return x

12.4.4.3 XML.prototype.appendChild (child)

Overview

The appendChild method appends a deep copy of the given child to the end of this XML object’s properties and returns this XML object. For example,

```javascript
var e = <employees>
  <employee id="0" ><name>Jim</name><age>25</age></employee>
  <employee id="1" ><name>Joe</name><age>20</age></employee>
</employees>;
// Add a new child element to the end of Jim's employee element
e.employee.(name == "Jim").appendChild(<hobby>snorkeling</hobby>);
```

Semantics
When the appendChild method of an XML object \( x \) is called with one parameter \( \text{child} \), the following steps are taken:

1. Let \( \text{children} \) be the result of calling the \([\text{Get}]\) method of \( x \) with argument "*"
2. Call the \([\text{Put}]\) method of \( \text{children} \) with arguments \( \text{children}[[\text{Length}]] \) and \( \text{child} \)
3. Return \( x \)

**12.4.4.4 XML.prototype.attribute ( attributeName )**

**Overview**

The attribute method returns an XMLList containing zero or one XML attributes associated with this XML object that have the given \( \text{attributeName} \). For example,

```javascript
// get the id of the employee named Jim
e.employee.(name == "Jim").attribute("id");
```

**Semantics**

When the attribute method of an XML object \( x \) is called with a parameter \( \text{attributeName} \), the following steps are taken:

1. Let \( \text{name} = \text{ToAttributeName}(\text{attributeName}) \)
2. Return the result of calling the \([\text{Get}]\) method of \( x \) with argument \( \text{name} \)

**12.4.4.5 XML.prototype.attributes ( )**

**Overview**

The attributes method returns an XMLList containing the XML attributes of this object. For example,

```javascript
// print the attributes of an XML object
function printAttributes(x) {
    for (var a in x.attributes()) {
        print("The attribute named " + a.name() + " has the value " + a);
    }
}
```

**Semantics**

When the attributes method of an XML object \( x \) is called, the following step is taken:

1. Return the result of calling the \([\text{Get}]\) method of \( x \) with argument \( \text{ToAttributeName("*")} \)

**12.4.4.6 XML.prototype.child ( propertyName )**

**Overview**

The child method returns the list of children in this XML object matching the given \( \text{propertyName} \). If \( \text{propertyName} \) is a numeric index, the child method returns a list containing the child at the ordinal position identified by \( \text{propertyName} \). For example,

```javascript
var name = customer.child("name"); // equivalent to: var name = customer.name;
var secondChild = customer.child(1); // equivalent to: var secondChild = customer.[1]
```

**Semantics**

When the child method of an XML object \( x \) is called, it performs the following step:
1. If ToString(ToUint32(P)) == P
   a. Let children be the result of calling the [[Get]] method of x with argument "*"
   b. Return the result of calling the [[Get]] method of children with argument propertyName
2. Return the result of calling the [[Get]] method of x with argument propertyName

12.4.4.7 XML.prototype.childIndex ( )

Overview

The childIndex method returns a Number representing the ordinal position of this XML object within the context of its parent. For example,

// Get the ordinal index of the employee named Joe.
var joeindex = e.employee.(name == "Joe").childIndex();

Semantics

When the childIndex method of an XML object x is called, it performs the following steps:

1. Let parent = x.\[
   \text{[[Parent]]}\]
2. If (parent == null) or (x.\[
   \text{[[Class]]}\] == "attribute"), return NaN
3. Let q be the property of parent, where parent[q] is the same object as x
4. Return ToNumber(q)

12.4.4.8 XML.prototype.children ( )

Overview

The children method returns an XMLList containing all the properties of this XML object in order. For example,

// Get child elements of first employee: returns an XMLList containing:
// <name>Jim</name>, <age>25</age> and <hobby>Snorkeling</hobby>
var emps = e.employee[0].children();

Semantics

When the children method of an XML object x is called, it performs the following step:

1. Return the results of calling the [[Get]] method of x with argument "*"

12.4.4.9 XML.prototype.comments ( )

Overview

The comments method returns an XMLList containing the properties of this XML object that represent XML comments.

Semantics

When the comments method of an XML object x is called, it performs the following steps:

1. Let l be a new XMLList with l.\[
   \text{[[TargetObject]]}\] = x and l.\[
   \text{[[TargetProperty]]}\] = null
2. For i = 0 to x.\[
   \text{[[Length]]}\]-1
   a. If x[i].\[
   \text{[[Class]]}\] == "comment", call the [[Append]] method of l with argument x[i]
3. Return l

12.4.4.10 XML.prototype.contains ( value )

Overview
The contains method returns the result of comparing this XML object with the given value. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value.

**Semantics**

When the contains method is called on an XML object \( x \) with parameter \( value \), the following step is taken:

1. Return the result of the comparison \( x == value \)

12.4.4.11 XML.prototype.copy ( )

**Overview**

The copy method returns a deep copy of this XML object with the internal [[Parent]] property set to null.

**Semantics**

When the copy method is called on an XML object \( x \), the following steps are taken:

1. Return the result of calling the [[DeepCopy]] method of \( x \)

12.4.4.12 XML.prototype.descendants ( [ name ] )

**Overview**

The descendants method returns all the XML valued descendants (children, grandchildren, great-grandchildren, etc.) of this XML object with the given name. If the name parameter is omitted, it returns all descendants of this XML object.

**Semantics**

When the descendants method is called on an XML object \( x \) with the optional parameter name, the following steps are taken:

1. If name is not specified, let name = "*"
2. Return the result of calling the [[Descendants]] method of \( x \) with argument name

12.4.4.13 XML.prototype.hasOwnProperty ( P )

**Overview**

The hasOwnProperty method returns a Boolean value indicating whether this object has the property specified by \( P \). For all XML objects except the XML prototype object, this is the same result returned by the internal method[[HasProperty]]. For the XML prototype object, hasOwnProperty also examines the list of local properties to determine if there is a method property with the given name.

**Semantics**

When the hasOwnProperty method of an XML object \( x \) is called with parameter \( P \), the following step is taken:

1. If the result of calling the [[HasProperty]] method of this object with argument \( P \) is \( \text{true} \), return \( \text{true} \)
2. If \( x \) has a property with name ToString(\( P \)), return \( \text{true} \)
3. Return \( \text{false} \)

12.4.4.14 XML.prototype.hasComplexContent( )

**Overview**

The hasComplexContent method returns a Boolean value indicating whether this XML object contains complex content. An XML object is considered to contain complex content if it represents an XML element that has child elements. XML objects representing attributes, comments, processing instructions and text nodes do not have complex content. The existence of
attributes, comments, processing instructions and text nodes within an XML object is not significant in determining if it has
complex content.

Semantics

When the hasComplexContent method is called on an XML object x, the following steps are taken:

1. If x.[[Class]] ∈ {"attribute", "comment", "processing-instruction", "text"}, return false
2. For each property p in x
   a. If x[p].[[Class]] === "element", return true
3. Return false

12.4.4.15 XML.prototype.hasSimpleContent( )

Overview

The hasSimpleContent method returns a Boolean value indicating whether this XML object contains simple content. An XML
object is considered to contain simple content if it represents a text node, represents an attribute node or if it represents an
XML element that has no child elements. XML objects representing comments and processing instructions do not have simple
content. The existence of attributes, comments, processing instructions and text nodes within an XML object is not significant
in determining if it has simple content.

Semantics

When the hasSimpleContent method is called on an XML object x, the following steps are taken:

1. If x.[[Class]] ∈ {"comment", "processing-instruction"}, return false
2. For each property p in x
   a. If x[p].[[Class]] === "element", return false
3. Return true

12.4.4.16 XML.prototype.inScopeNamespaces( )

Overview

The inScopeNamespaces method returns an Array of Namespace objects representing the namespaces in scope for this XML
object in the context of its parent. If the parent of this XML object is modified, the associated namespace declarations may
change.

Semantics

When the inScopeNamespaces method is called on an XML object x, the following steps are taken:

1. Let y = x
2. Let inScopeNS = { }
3. While (y is not null)
   a. For each ns in y.[[InScopeNamespaces]]
      i. If ns.prefix is undefined
         or there exists no namespace n ∈ inScopeNS, such that n.prefix === ns.prefix
            1. Let inScopeNS = inScopeNS ∪ { ns }
   b. Let y = y.[[Parent]]
4. Note: The E4X data model does not enforce the constraint: ∀ x ∈ XML : x.[[InScopeNamespaces]] ⊇
x.[[Parent]].[[InScopeNamespaces]]. However, implementations that do enforce this constraint may set inScopeNS =
x.[[InScopeNamespaces]] instead of using the computation above.
5. Let a be a new Array created as if by calling the constructor, new Array()
6. Let i = 0
7. For each ns in inScopeNS
   a. Call the [[Put]] method of a with arguments ToString(i) and ns
   b. Let i = i + 1
12.4.4.17 XML.prototype.insertChildAfter (child1, child2)

Overview

The insertChildAfter method inserts the given child2 after the given child1 in this XML object and returns this XML object. If child1 is null, the insertChildAfter method inserts child2 before all children of this XML object (i.e., after none of them). If child1 does not exist in this XML object, it returns without modifying this XML object.

Semantics

When the insertChildAfter method is called on an XML object x with parameters child1 and child2, the following steps are taken:

1. If x.[[Class]] ∈ \{"text", "comment", "processing-instruction", "attribute"\}, return
2. If (child1 == null)
   a. Call the [[Insert]] method of x with arguments "0" and child2
   b. Return x
3. Else if Type(child1) is XML
   a. For i = 0 to x.[[Length]]-1
      i. If x[i] is the same object as child1
         1. Call the [[insert]] method of x with arguments ToString(i + 1) and child2
         2. Return x
4. Return

12.4.4.18 XML.prototype.insertChildBefore (child1, child2)

Overview

The insertChildBefore method inserts the given child2 before the given child1 in this XML object and returns this XML object. If child1 is null, the insertChildBefore method inserts child2 after all children in this XML object (i.e., before none of them). If child1 does not exist in this XML object, it returns without modifying this XML object.

Semantics

When the insertChildBefore method is called on an XML object x with parameters child1 and child2, the following steps are taken:

1. If x.[[Class]] ∈ \{"text", "comment", "processing-instruction", "attribute"\}, return
2. If (child1 == null)
   a. Call the [[Insert]] method of x with arguments ToString(x.[[Length]]) and child2
   b. Return x
3. Else if Type(child1) is XML
   a. For i = 0 to x.[[Length]]-1
      i. If x[i] is the same object as child1
         1. Call the [[insert]] method of x with arguments ToString(i) and child2
         2. Return x
4. Return

12.4.4.19 XML.prototype.length ()

Overview

The length method always returns the integer 1 for XML objects. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value to simplify the programmer’s task.

Semantics

When the length method is called on an XML object x, the following step is taken:
2. Return 1

12.4.4.20 XML.prototype.localName ( )

Overview
The localName method returns the local name portion of the qualified name of this XML object.

Semantics
When the localName method is called on an XML object x, the following step is taken:

1. If x.[[Name]] == null, return null
2. Return x.[[Name]].localName

12.4.4.21 XML.prototype.name ( )

Overview
The name method returns the qualified name associated with this XML object.

Semantics
When the name method is called on an XML object x, the following step is taken:

1. Return x.[[Name]]

12.4.4.22 XML.prototype.namespace ( [ prefix ] )

Overview
If no prefix is specified, the namespace method returns the Namespace associated with the qualified name of this XML object.

If a prefix is specified, the namespace method looks for a namespace in scope for this XML object with the given prefix and, if found, returns it. If no such namespace is found, the namespace method returns undefined.

Semantics
When the namespace method is called on an XML object x with zero arguments or one argument prefix, the following steps are taken:

2. Let y = x
3. Let inScopeNS = { }
4. While (y is not null)
   a. For each ns in y.[[InScopeNamespaces]]
      i. If ns.prefix is undefined
         or there exists no namespace n ∈ inScopeNS, such that n.prefix == ns.prefix
            1. Let inScopeNS = inScopeNS ∪ { ns }
   b. Let y = y.[[Parent]]
5. Note: The E4X data model does not enforce the constraint: ∀ x ∈ XML : x.[[InScopeNamespaces]] ⊇ x.[[Parent]].[[InScopeNamespaces]]. However, implementations that do enforce this constraint may set inScopeNS = x.[[InScopeNamespaces]] instead of using the computation above.
6. If prefix was not specified
   a. If x.[[Class]] ∈ { "text", "comment", "processing-instruction" }, return null
   b. Return the result of calling the [[GetNamespace]] method of x.[[Name]] with argument inScopeNS
7. Else
   a. Let prefix = ToString(prefix)
   b. Find a Namespace ns ∈ inScopeNS, such that ns.prefix = prefix. If no such ns exists, let ns = undefined.
   c. Return ns
12.4.4.23 XML.prototype.namespaceDeclarations ( )

Overview

The namespaceDeclarations method returns an Array of Namespace objects representing the namespace declarations associated with this XML object in the context of its parent. If the parent of this XML object is modified, the associated namespace declarations may change.

Semantics

When the namespaceDeclarations method is called on an XML object x, the following steps are taken:

1. Let a be a new Array created as if by calling the constructor, new Array()
2. If x.[[Class]] ∈ {"text", "comment", "processing-instruction", "attribute"}, return a
3. Let y = x.[[Parent]]
4. Let ancestorNS = { }
5. While (y is not null)
   a. For each ns in y.[[InScopeNamespaces]]
      i. If ns.prefix is undefined
         or there exists no namespace n ∈ ancestorNS, such that n.prefix == ns.prefix
         i. Let ancestorNS = ancestorNS ∪ { ns }
   b. Let y = y.[[Parent]]
6. Note: The E4X data model does not enforce the constraint: ∀ x ∈ XML : x.[[InScopeNamespaces]] ⊇ x.[[Parent]].[[InScopeNamespaces]]. However, implementations that do enforce this constraint may set ancestorNS = x.[[Parent]].[[InScopeNamespaces]] instead of using the computation above.
7. Let declaredNS = { }
8. for each ns in x.[[InScopeNamespaces]]
   a. if there exists no namespace n ∈ ancestorNS, such that n.prefix == ns.prefix and n.uri == ns.uri
      i. Let declaredNS = declaredNS ∪ { ns }
9. Let i = 0
10. For each ns in declaredNS
    a. Call the [[Put]] method of a with arguments ToString(i) and ns
    b. Let i = i + 1
11. Return a

12.4.4.24 XML.prototype.nodeKind ( )

Overview

The nodeKind method returns a string representing the [[Class]] of this XML object.

Semantics

When the nodeKind method is called on an XML object x, the following step is taken:

1. Return x.[[Class]]

12.4.4.25 XML.prototype.normalize ( )

Overview

The normalize method puts all text nodes in this and all descendant XML objects into a normal form by merging adjacent text nodes and eliminating empty text nodes. It returns this XML object.

Semantics

When the normalize method is called on an XML object x, the following steps are taken:

1. Let i = 0
2. While $i < x.\text{[Length]}$
   a. If $x[i].\text{[Class]} =$ "element"
      i. Call the normalize method of $x[i]$
      ii. Let $i = i + 1$
   b. Else if $x[i].\text{[Class]} =$ "text"
      i. While ($i + 1 < x.\text{[Length]}$) and ($x[i+1].\text{[Class]} =$ "text")
         1. Let $x[i].\text{[Value]}$ be the result of concatenating $x[i].\text{[Value]}$ and $x[i+1].\text{[Value]}$
         2. Call the $\text{[DeleteByIndex]}$ method of $x$ with argument $\text{ToString}(i + 1)$
      ii. If $x[i].\text{[Value]}$.length $\text{===}$ 0
         1. Call the $\text{[DeleteByIndex]}$ method of $x$ with argument $\text{ToString}(i)$
      iii. Else
         1. Let $i = i + 1$
   c. Else
      i. Let $i = i + 1$

3. Return $x$

12.4.4.26 XML.prototype.parent ( )

Overview

The parent method returns the parent of this XML object. For example,

```javascript
// Get the parent element of the second name in "e". Returns <employee id="1" …
var firstNameParent = e..name[1].parent();
```

Semantics

When the parent method is called on an XML object $x$, the following step is taken:

1. Return $x.\text{[Parent]}$

12.4.4.27 XML.prototype.processingInstructions ( [ name ] )

Overview

When the processingInstructions method is called with one parameter name, it returns an XMLList containing all the children of this XML object that are processing-instructions with the given name. When the processingInstructions method is called with no parameters, it returns an XMLList containing all the children of this XML object that are processing-instructions regardless of their name.

Semantics

When the processingInstructions method is called on an XML object $x$ with optional parameter name, the following steps are taken:

1. If name is not specified, let name = "*"
2. Let name = ToXMLName(name)
3. Let $l$ = a new XMLList with $l.\text{[TargetObject]} = x$ and $l.\text{[TargetProperty]} = \text{null}$
4. For $i = 0$ to $x.\text{[Length]}$-1
   a. If $x[i].\text{[Class]} =$ "processing-instruction"
      i. If name.localName == "*" or name.localName == $x[i].\text{[Name]}$.localName
         1. Call the $\text{[Append]}$ method of $l$ with argument $x[i]$
5. Return $l$

12.4.4.28 XML.prototype.prependChild ( value )

Overview
The prependChild method inserts a deep copy of the given child into this object prior to its existing XML properties and returns this XML object. For example,

```xml
// Add a new child element to the front of John's employee element
e.employee.(name == "John").prependChild(<prefix>Mr.</prefix>);
```

**Semantics**

When the prependChild method is called on an XML object \( x \) with parameter \( value \), the following steps are taken:

1. Call the \([[\text{Insert}]\]) method of this object with arguments "0" and \( value \)
2. Return \( x \)

**12.4.4.29 XML.prototype.propertyIsEnumerable ( P )**

**Overview**

The propertyIsEnumerable method returns a Boolean value indicating whether the property \( P \) is accessible via the XML \([[\text{Get}]\]) method. All properties of XML objects accessible via the XML \([[\text{Get}]\]) method are enumerable.

**Semantics**

When the propertyIsEnumerable method of an XML object \( x \) is called with parameter \( P \), the following step is taken:

1. Return the result of calling the \([[\text{HasProperty}]\]) method of this object with argument \( P \).

**12.4.4.30 XML.prototype.removeNamespace ( namespace )**

**Overview**

The removeNamespace method removes the given namespace from the in scope namespaces of this object and all its descendants, then returns a copy of this XML object. The removeNamespaces method will not remove a namespace from an object where it is referenced by that object’s QName or the QNames of that object’s attributes.

**Semantics**

When the removeNamespace method is called on an XML object \( x \) with parameter \( namespace \), the following steps are taken:

2. If \( x.[[\text{Class}]] \) \in \{"text", "comment", "processing-instruction", "attribute"\}, return \( x \)
3. Let \( ns \) be a Namespace object created as if by calling the function Namespace( \( namespace \) )
4. Let \( thisNS \) be the result of calling \([[\text{GetNamespace}]\]) on \( x.[[\text{Name}]] \) with argument \( x.[[\text{InScopeNamespaces}]] \)
5. If \( (thisNS === ns) \), return \( x \)
6. For each attribute \( a \) in \( x.[[\text{Attributes}]] \)
   a. Let \( aNS \) be the result of calling \([[\text{GetNamespace}]\]) on \( a.[[\text{Name}]] \) with argument \( x.[[\text{InScopeNamespaces}]] \)
   b. If \( (aNS === ns) \), return \( x \)
7. If \( ns.prefix === \text{undefined} \)
   a. If there exists a namespace \( n \in x.[[\text{InScopeNamespaces}]] \), such that \( n.uri === ns.uri \), remove the namespace \( n \) from \( x.[[\text{InScopeNamespaces}]] \)
8. else
   a. If there exists a namespace \( n \in x.[[\text{InScopeNamespaces}]] \), such that \( n.uri === ns.uri \) and \( n.prefix === ns.prefix \), remove the namespace \( n \) from \( x.[[\text{InScopeNamespaces}]] \)
9. For each property \( p \) of \( x \)
   a. If \( p.[[\text{Class}]] === \text{"element"} \), call the removeNamespace method of \( p \) with argument \( ns \)
10. Note: The E4X data model does not enforce the constraint: \( \forall x \in \text{XML} : x.[[\text{InScopeNamespaces}]] \supseteq x.[[\text{Parent}]].[[\text{InScopeNamespaces}]]. \) However, implementations may at this point remove namespace from the \([[\text{InScopeNamespaces}]]\) property of any ancestors of \( x \) provided \( namespace \) is not referenced by the associated ancestor's QName or the QNames of the associated ancestor's attributes.
11. Return \( x \)
12.4.4.31  XML.prototype.replace ( propertyName, value )

Overview

The replace method replaces the XML properties of this XML object specified by propertyName with value and returns this XML object. If this XML object contains no properties that match propertyName, the replace method returns without modifying this XML object. The propertyName parameter may be a numeric property name, an unqualified name for a set of XML elements, a qualified name for a set of XML elements or the properties wildcard “*”. When the propertyName parameter is an unqualified name, it identifies XML elements in the default namespace. The value parameter may be an XML object, XMLList object or any value that may be converted to a String with ToString(). For example,

```
// Replace the first employee record with an open staff requisition
employees.replace(0, <requisition status="open"/>);

// Replace all item elements in the order with a single empty item
order.replace("item", <item/>);
```

Semantics

When the replace method is called on an XML object x with parameters propertyName and value, the following steps are taken:

1. If x.[[Class]] ∈ {"text", "comment", "processing-instruction", "attribute"}, return x
2. If Type(value) ≠ {XML, XMLList}, let c = ToString(value)
3. Else let c be the result of calling the [[DeepCopy]] method of value
4. If ToString(ToUint32(P)) === P
   a. Call the [[Replace]] method of x with arguments P and c and return x
5. Let n be a QName object created as if by calling the function QName(P)
6. Let i = undefined
7. For k = 0 to x.[[Length]]-1
   a. If ((n.localName === "*")
      or ((s[k].[[Class]] === "element") and (s[k].[[Name]].localName==n.localName))
      and ((n.uri == null) or (n.uri === x[k].[[Name]].uri ))
      i. If (i === undefined), let i = k
      ii. Else call the [[DeleteByIndex]] method of x with argument ToString(k)
8. If i === undefined, return x
9. Call the [[Replace]] method of x with arguments ToString(i) and c
10. Return x

12.4.4.32  XML.prototype.setChildren ( value )

Overview

The setChildren method replaces the XML properties of this XML object with a new set of XML properties from value. value may be a single XML object or an XMLList. setChildren returns this XML object. For example,

```
// Replace the entire contents of Jim's employee element
e.employee.(name == "Jim").setChildren(<name>John</name> + <age>35</age>);
```

Semantics

When the setChildren method is called on an XML object x with parameter value, the following steps are taken:

1. Call the [[Put]] method of x with arguments "*" and value
2. Return x
12.4.4.33  XML.prototype.setLocalName ( name )

Overview

The setLocalName method replaces the local name of this XML object with a string constructed from the given name.

Semantics

When the setLocalName method is called on an XML object x with parameter name, the following steps are taken:

1. If x.[[Class]] ∈ {"text", "comment"}, return
2. if (Type(name) is Object) and (name.[[Class]] == "QName")
   a. Let name = name.localName
3. else
   a. Let name = ToString(name)
4. Let x.[[Name]].localName = name

12.4.4.34  XML.prototype.setName ( name )

Overview

The setName method replaces the name of this XML object with a QName or AttributeName constructed from the given name.

Semantics

When the setName method is called on an XML object x with parameter name, the following steps are taken:

1. If x.[[Class]] ∈ {"text", "comment"}, return
2. if (Type(name) is Object) and (name.[[Class]] == "QName") and (name.uri == null)
   a. Let name = name.localName
3. Let n be a new QName created if by calling the constructor new QName(name)
4. if x.[[Class]] == "processing-instruction", let n.uri be the empty string
5. Let x.[[Name]] = n

12.4.4.35  XML.prototype.setNamespace ( ns )

Overview

The setNamespace method replaces the namespace associated with the name of this XML object with the given namespace.

Semantics

When the setNamespace method is called on an XML object x with parameter ns, the following step is taken:

1. If x.[[Class]] ∈ {"text", "comment", "processing-instruction"}, return
2. Let namespace be a new Namespace created as if by calling the constructor new Namespace(ns)
3. Let x.[[Name]] be a new QName created as if by calling the constructor new QName(namespace, x.[[Name]])

12.4.4.36  XML.prototype.text ( )

Overview

The text method returns an XMLList containing all XML properties of this XML object that represent XML text nodes.

Semantics

When the text method of an XML object x is called, the following steps are taken:

1. Let l be a new XMLList with l.[[TargetObject]] = x and l.[[TargetProperty]] = null
2. For \( i = 0 \) to \( x.([\text{Length}])-1 \)
   a. If \( x[i].([\text{Class}]) == \text{"text"} \), Call the \([\text{Append}]\) method of \( l \) with argument \( x[i] \)
3. Return \( l \)

12.4.4.37 \ xml.prototype.toString ( )

Overview

The toString method returns a string representation of this XML object per the ToString conversion operator described in section 9.1.

Semantics

When the toString method of an XML object \( x \) is called, the following step is taken:

1. Return ToString\( (x) \)

12.4.4.38 \ xml.prototype.toXMLString ( )

Overview

The toXMLString() method returns an XML encoded string representation of this XML object per the ToXMLString conversion operator described in section 9.2. Unlike the toString method, toXMLString provides no special treatment for XML objects that contain only XML text nodes (i.e., primitive values). The toXMLString method always includes the start tag, attributes and end tag of the XML object regardless of its content. It is provided for cases when the default XML to string conversion rules are not desired. For example,

```
var item = <item>
    <description>Laptop Computer</description>
    <price>2799.95</price>
    <quantity>1</quantity>
</item>;
// returns "Description stored as <description>Laptop Computer</description>"
var logmsg = "Description stored as " + item.description.toXMLString();
// returns "Thank you for purchasing a Laptop Computer!" (with tags removed)
var message = "Thank you for purchasing a " + item.description + "!";
```

Semantics

When the toXMLString method of an XML object \( x \) is called, the following step is taken:

1. Return ToXMLString\( (x) \)

12.4.4.39 \ xml.prototype.valueOf ( )

Overview

The valueOf method returns this XML object.

Semantics

When the valueOf method of an XML object \( x \) is called, the following step is taken:

1. Return \( x \)
12.4.5 Properties of XML Instances
XML instances have no special properties beyond those inherited from the XML prototype object.

12.5 XMLList Objects

12.5.1 The XMLList Constructor Called as a Function

Syntax

XMLList(value)

Overview

When XMLList is called as a function rather than as a constructor, it converts its argument to an XMLList object. When its argument is an XMLList, the input argument is returned without modification.

Semantics

When XMLList is called as a function with parameter value, the following steps are taken:

1. If value is null, undefined or not supplied, let value be the empty string
2. Return ToXMLList(value)

12.5.2 The XMLList Constructor

Syntax

new XMLList([value])

Overview

When XMLList is called as part of a new expression, it is a constructor and creates a new XMLList object. When the XMLList constructor is called with no arguments, it returns an empty XMLList. When the XMLList constructor is called with a value of type XMLList, the XMLList constructor returns a shallow copy of the value. When the XMLList constructor is called with a non-XMLList value, it converts its input argument to an XMLList object.

Semantics

When the XMLList constructor is called with an optional parameter value, the following steps are taken:

1. if value is null, undefined or not supplied, let value be the empty string
2. If Type(value) is XMLList
   a. Let l be a new XMLList object with l.([TargetObject]) = null
   b. Call the [[Append]] method of l with argument value
   c. Return l
3. Else
   a. Return ToXMLList(value)

12.5.3 Properties of the XMLList Constructor

The value of the internal [[Prototype]] property of the XMLList constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 1), the XMLList constructor has the following properties:

12.5.3.1 XMLList.prototype

The initial value of the XMLList.prototype property is the XMLList prototype object (section 12.5.4).
This property has the attributes \{ DontEnum, DontDelete, ReadOnly \}.

12.5.4 Properties of the XMLList Prototype Object (Built-in Methods)

Each value of type XMLList has a set of built-in methods available for performing common operations. These built-in methods are described in the following sections.

The XMLList prototype object is itself an XMLList object (its [[Class]] property is "XMLList") whose value is the empty XMLList.

The value of the internal [[Prototype]] property of the XMLList prototype object is the Object prototype object. (section 15.2.3.1 of ECMAScript Edition 3).

Note: To simplify the programmer’s task, E4X intentionally blurs the distinction between a single XML object and an XMLList containing only one value. To this end, E4X extends the ECMAScript function call semantics such that all methods available for values of type XML are also available for XMLLists of size one. See section 10.2.2 for more information.

None of the built-in functions defined on XMLList.prototype are generic. They throw a TypeError exception if the this value is not an XMLList object. Therefore, they cannot be transferred to other kinds of objects for use as a method.

12.5.4.1 XMLList.prototype.constructor

The initial value of the XMLList prototype constructor is the built-in XMLList constructor.

12.5.4.2 XMLList.prototype.attribute (attributeName)

Overview

The attribute method calls the attribute method of each XML object in this XMLList object passing attributeName as a parameter and returns an XMLList containing the results in order.

Semantics

When the attribute method is called on an XMLList object \( l \) with parameter attributeName, the following steps are taken:

1. Let name = ToAttributeName(attributeName)
2. Return the result of calling the [[Get]] method of \( l \) with argument name

12.5.4.3 XMLList.prototype.attributes ()

Overview

The attributes method calls the attributes() method of each XML object in this XMLList object and returns an XMLList containing the results in order.

Semantics

When the attributes method is called on an XMLList object \( l \), the following step is taken:

1. Return the result of calling the [[Get]] method of \( l \) with argument ToAttributeName("*")

12.5.4.4 XMLList.prototype.child (propertyName)

Overview

The child method calls the child() method of each XML object in this XMLList object and returns an XMLList containing the results in order.

Semantics

When the child method is called on an XMLList object \( l \) with parameter propertyName, the following step is taken:
1. Let $m$ be a new XMLList with $m.[[TargetObject]] = l$.
2. For $i = 0$ to $l.[[Length]]-1$
   a. Let $r = l[i].child(propertyName)$
   b. If $r.[[Length]] > 0$, call the [[Append]] method of $m$ with argument $r$

12.5.4.5 XMLList.prototype.children ( )

Overview

The children method calls the children() method of each XML object in this XMLList object and returns an XMLList containing the results concatenated in order. For example,

```plaintext
// get all the children of all the items in the order
var allitemchildren = order.item.children();

// get all grandchildren of the order that have the name price
var grandChildren = order.children().price;
```

Semantics

When the children method is called on an XMLList object $l$, the following step is taken:

1. Return the results of calling the [[Get]] method of $l$ with argument "*"

12.5.4.6 XMLList.prototype.comments ( )

Overview

The comments method calls the comments method of each XML object in this XMLList object and returns an XMLList containing the results concatenated in order.

Semantics

When the comments method is called on an XMLList object $l$, the following steps are taken:

1. Let $m$ be a new XMLList with $m.[[TargetObject]] = l$
2. For $i = 0$ to $l.[[Length]]-1$
   a. If $l[i].[[Class]] == "element"
      i. Let $r = l[i].comments()$
   b. If $r.[[Length]] > 0$, call the [[Append]] method of $m$ with argument $r$
3. Return $m$

12.5.4.7 XMLList.prototype.contains ( value )

Overview

The contains method returns a boolean value indicating whether this XMLList object contains an XML object that compares equal to the given $value$.

Semantics

When the contains method is called on an XMLList object $l$ with parameter $value$, the following steps are taken:

1. For $i = 0$ to $l.[[Length]]-1$
   a. If the result of the comparison $l[i] == value$ is true, return true
2. Return false
12.5.4.8 XMLList.prototype.copy ( )

Overview

The copy method returns a deep copy of this XMLList object.

Semantics

When the copy method is called on an XMLList object \( l \), the following step is taken:

1. Return the result of calling the [[DeepCopy]] method of \( l \)

12.5.4.9 XMLList.prototype.descendants ( [ name ] )

The descendants method calls the descendants method of each XML object in this XMLList object with the optional parameter \( name \) (or the string "*" if \( name \) is omitted) and returns an XMLList containing the results concatenated in order.

Semantics

When the descendants method is called on an XMLList object \( l \) with optional parameter \( name \), the following steps are taken:

1. If \( name \) is not specified, \( name = "*" \)
2. Return the result of calling the [[Descendants]] method of \( l \) with argument \( name \)

12.5.4.10 XMLList.prototype.hasOwnProperty ( P )

Overview

The hasOwnProperty method returns a Boolean value indicating whether this object has the property specified by \( P \). For all XMLList objects except the XMLList prototype object, this is the same result returned by the internal method[[HasProperty]]. For the XMLList prototype object, hasOwnProperty also examines the list of local properties to determine if there is a method property with the given name.

Semantics

When the hasOwnProperty method of an XMLList object \( x \) is called with parameter \( P \), the following step is taken:

1. If the result of calling the [[HasProperty]] method of this object with argument \( P \) is \( \text{true} \), return \( \text{true} \)
2. If \( x \) has a property with name ToString(\( P \)), return \( \text{true} \)
3. Return \( \text{false} \)

12.5.4.11 XMLList.prototype.hasComplexContent( )

Overview

The hasComplexContent method returns a Boolean value indicating whether this XMLList object contains complex content. An XMLList object is considered to contain complex content if it is not empty, contains a single XML item with complex content or contains elements.

Semantics

When the hasComplexContent method is called on an XMLList object \( x \), the following steps are taken:

1. If \( x.\text{[[Length]]} = 0 \), return \( \text{false} \)
2. If \( x.\text{[[Length]]} = 1 \), return \( x[0]\).hasComplexContent()
3. For each property \( p \) in \( x \)
   a. If \( x[p].\text{[[Class]]} = \text{"element"} \), return \( \text{true} \)
4. Return \( \text{false} \)
### 12.5.4.12 XMLList.prototype.hasSimpleContent( )

**Overview**

The hasSimpleContent method returns a Boolean value indicating whether this XMLList contains simple content. An XMLList object is considered to contain simple content if it is empty, contains a single XML item with simple content or contains no elements.

**Semantics**

When the hasSimpleContent method is called on an XMLList object `x`, the following steps are taken:

1. If `x.[[Length]] == 0`, return `true`
2. If `x.[[Length]] == 1`, return `x[0].hasSimpleContent()`
3. For each property `p` in `x`
   a. If `x[p].[[Class]] == "element"`, return `false`
4. Return `true`

### 12.5.4.13 XMLList.prototype.length ( )

**Overview**

The length method returns the number of properties in this XMLList object. For example,

```javascript
for (var i = 0; i < e..name.length(); i++) {
    print("Employee name:" + e..name[i]);
}
```

**Semantics**

When the length method of an XMLList object `l` is called, the following step is taken:

1. Return `l.[[Length]]`

### 12.5.4.14 XMLList.prototype.normalize ( )

**Overview**

The normalize method puts all text nodes in this XMLList, all the XML objects it contains and the descendents of all the XML objects it contains into a normal form by merging adjacent text nodes and eliminating empty text nodes. It returns this XMLList object.

**Semantics**

When the normalize method is called on an XMLList object `l`, the following steps are taken:

1. Let `i = 0`
2. While `i < l.[[Length]]`
   a. If `l[i].[[Class]] == "element"`
      i. Call the normalize method of `l[i]`
      ii. Let `i = i + 1`
   b. Else if `l[i].[[Class]] == "text"`
      i. While `(i+1) < l.[[Length]]` and `(l[i+1].[[Class]] == "text")`
         1. Let `l[i].[[Value]]` be the result of concatenating `l[i].[[Value]]` and `l[i+1].[[Value]]`
         2. Call the [[Delete]] method of `l` with argument ToString(`i + 1`)
      ii. If `l[i].[[Value]].length == 0`
         1. Call the [[Delete]] method of `l` with argument ToString(`i`)
      iii. Else
         1. Let `i = i + 1`
c. Else
   i. Let \( i = i + 1 \)
3. Return \( l \)

### 12.5.4.15 XMLList.prototype.parent ( )

**Overview**

If all items in this XMLList object have the same parent, it is returned. Otherwise, the parent method returns `undefined`.

**Semantics**

When the parent method is called on an XMLList object \( l \), the following steps are taken:

1. If \( l.\text{[[Length]]} = 0 \), return `undefined`
2. Let \( \text{parent} = l.\text{[[0]].[[Parent]]} \)
3. For \( i = 1 \) to \( l.\text{[[Length]]}-1 \), if \( l[i].\text{[[Parent]]} \) is not equal to \( \text{parent} \), return `undefined`
4. Return \( \text{parent} \)

### 12.5.4.16 XMLList.prototype.processingInstructions ( [ name ] )

**Overview**

The processingInstructions method calls the processingInstructions method of each XML object in this XMLList object passing the optional parameter `name` (or `"*"` if it is omitted) and returns an XMLList containing the results in order.

**Semantics**

When the processingInstructions method is called on an XMLList object \( l \) with optional parameter `name`, the following steps are taken:

1. If `name` is not specified, let `name = "*"`
2. Let `name = ToXmlName(name)`
3. Let \( m = \text{a new XMLList with } m.\text{[[TargetObject]]} = l \)
4. For \( i = 0 \) to \( l.\text{[[Length]]}-1 \)
   a. If \( l[i].\text{[[Class]]} \) == "element"
      i. Let \( r = l[i].\text{processingInstructions(name)} \)
      ii. If \( r.\text{[[Length]]} > 0 \), call the \([\text{Append}]\) method of \( m \) with argument \( r \)
5. Return \( m \)

### 12.5.4.17 XMLList.prototype.propertyIsEnumerable ( P )

**Overview**

The propertyIsEnumerable method returns a Boolean value indicating whether the property \( P \) is accessible via the XMLList \([\text{Get}]\) method. All properties of XMLList objects accessible via the XMLList \([\text{Get}]\) method are enumerable.

**Semantics**

When the propertyIsEnumerable method of an XMLList object \( x \) is called with parameter \( P \), the following step is taken:

1. Return the result of calling the \([\text{HasProperty}]\) method of this object with argument \( P \).

### 12.5.4.18 XMLList.prototype.text ( )

**Overview**

The text method calls the text method of each XML object contained in this XMLList object and returns an XMLList containing the results concatenated in order.
Semantics

When the text method is called on an XMLList object \( l \), the following steps are taken:

1. Let \( m \) be a new XMLList with \( m.\text{[TargetObject]} = l \)
2. For \( i = 0 \) to \( l.\text{[Length]} - 1 \)
   a. If \( l[i].\text{[Class]} == \text{"element"} \)
      i. Let \( r = l[i].\text{text()} \)
      ii. If \( r.\text{[Length]} > 0 \), call the \([\text{Append}]\) method of \( m \) with argument \( r \)
3. Return \( m \)

12.5.4.19 XMLList.prototype.toString()

Overview

The toString method returns a string representation of this XMLList object per the ToString conversion operator described in section 9.1.

Semantics

When the toString() method of an XMLList object \( l \) is called, the following step is taken:

1. Return ToString(\( l \))

12.5.4.20 XMLList.prototype.toXMLString()

Overview

The toXMLString() method returns an XML encoded string representation of this XMLList object per the ToXMLString conversion operator described in section 9.2. Unlike the toString method, toXMLString provides no special treatment for XML objects that contain only XML text nodes (i.e., primitive values). The toXMLString method always calls toXMLString on each property contained within this XMLList object, concatenates the results in order and returns a single string.

Semantics

When the toXMLString() method of an XMLList object \( l \) is called, the following step is taken:

1. Return toXMLString(\( l \))

12.5.4.21 XMLList.prototype.valueOf()

Overview

The valueOf method returns this XMLList object.

Semantics

When the valueOf method of an XMLList object \( l \) is called, the following step is taken:

1. Return \( l \)
13 Errors

E4X extends the list of errors implementations are not required to report as specified as follows:

- An implementation may define behaviour other than throwing a `TypeError` exception for the `ToXML` function and `ToXMLList` function when they are called with an argument of type `Object`. 
A  Optional Features (Non-normative)

This section describes optional E4X features that are not required to be conformant with E4X, but may be provided by a conforming E4X implementation. Implementations that implement these features should conform to the associated specifications provided herein.

A.1 XML Built-in Methods

An E4X implementation may add the following methods to XML objects

A.2.1 domNode( )

Overview

The domNode method returns a W3C DOM Node representation of this XML Object.

Semantics

The semantics of the domNode method are implementation dependent.

A.2.2 domNodeList( )

Overview

The domNodeList method returns a W3C DOM NodeList containing a single W3C DOM Node representation of this XML Object.

Semantics

The semantics of the domNodeList method are implementation dependent.

A.2.3 xpath (XPathExpression)

Overview

The xpath method evaluates the XPathExpression in accordance with the W3C XPath Recommendation using this XML object as the context node. Before evaluating the XPathExpression, the xpath method sets the XPath context as follows. The context node is set to this XML object. The context position is set to 1. The context size is set to 1. The set of variable bindings is set to the empty set. The function library is set to the empty set. The set of namespaces is set according to the set of in scope namespaces on this XML object. If the XPathExpression evaluates to a node list, the xpath method returns the results as an XMLList. Otherwise, the xpath method throws a TypeError exception. For example,

```javascript
// Use an xpath expression to get all the employees named John Smith
var jim = e.xpath("//employee[name='John Smith']")
```

Semantics

When the xpath method of an XML object $x$ is called with parameter $XPathExpression$ it performs the following steps:

1. Let $e$ be the result of evaluating $XPathExpression$
2. Let $s = $ToString($GetValue(XpathExpression))$
3. Create an XPath context object representing the XML object $x$. This semantics of this step are implementation dependent.
4. Let the XPath context position be 1
5. Let the XPath context size be 1
6. Let the XPath variable bindings be $\{\}$
7. Let the XPath function library be $\{\}$
8. Let the XPath namespaces be $x.\{[InScopeNamespaces]\}$
9. Let $r$ be the result of evaluating $s$ in accordance with the W3C XPath Recommendation
10. If $r$ is an XPath node-set, convert it to an XMLList in an implementation dependent way and return it.
11. Throw a TypeError exception
A.2 XMLList Built-in Methods

An E4X implementation may add the following methods to XMLList objects

A.2.4 domNode( )

Overview

If this XMLList object contains only one XML value \( x \), the domNode method returns a W3C DOM Node representation of \( x \). Otherwise, the domNode method returns undefined.

Semantics

The semantics of the domNode method are implementation dependent.

A.3.1 domNodeList( )

Overview

The domNodeList method returns a W3C DOM NodeList representation of this XMLList Object.

Semantics

The semantics of the domNodeList method are implementation dependent.

A.3.2 xpath (XPathExpression)

Overview

The xpath method evaluates the XPathExpression for each XML property contained in this XMLList object and concatenates the results an XMLList containing the results concatenated in order.

Semantics

When the xpath method is called on an XMLList object \( l \) with parameter \( XPathExpression \), the following steps are taken:

1. Let \( m = \) a new XMLList with \( l.\text{[[TargetObject]]} = \text{null} \)
2. For \( i = 0 \) to \( l.\text{[[Length]]} - 1 \)
   a. If \( \text{Type}(l[i]) \) is XML and \( l[i].\text{[[Class]]} = \text{"element"} \)
      i. Let \( r = l[i].\text{xpath(XPathExpression)} \)
      ii. If \( r.\text{[[Length]]} > 0 \), call the [[Append]] method of \( m \) with argument \( r \)
3. Return \( m \)
Free printed copies can be ordered from:

**ECMA**
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax: +41 22 849.60.01
Email: documents@ecma.ch

Files of this Standard can be freely downloaded from the ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.